O'REILLY®




图灵程序设计丛书



# HTML5 数据推送应用开发

Data Push Apps with HTML5 SSE

[美] Darren Cook 著
刘帅 译

人民邮电出版社
POSTS & TELECOM PRESS

# 数字版权声明

## 译者介绍



### 刘帅

百度前端高级研发工程师，毕业于哈尔滨工程大学，获得计算机科学与技术专业学士学位。毕业以来一直从事前端开发工作，先后就职于新浪、阿里巴巴、腾讯、百度，参与开发了基于HTML5技术的腾讯浏览器、QQ for Windows 8、百度打车WebApp版，现参与开发百度地图。


图灵程序设计丛书

# HTML5数据推送应用开发

## Data Push Apps with HTML5 SSE


[美] Darren Cook 著

刘帅 译



O'REILLY®

*Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo*

O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社

北　　京

图书在版编目（CIP）数据

HTML5数据推送应用开发 / （美）库克（Cook,D.）著；
刘帅译. -- 北京 : 人民邮电出版社, 2014.11
（图灵程序设计丛书）
ISBN 978-7-115-37059-4

Ⅰ. ①H… Ⅱ. ①库… ②刘… Ⅲ. ①超文本标记语言
—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第212735号

## 内 容 提 要

本书详细介绍了SSE（Server-Sent Event，服务端推送事件）。SSE是一种允许服务端向客户端推送新数据的HTML5技术。利用这种技术，网页可以迅速加载，并且能及时获得用户感兴趣的最新数据。相比数据拉取，SSE是更优的解决方案，能最大限度地降低延迟。本书通过丰富的示例详细叙述了SSE的优势、它在的日常生活中的应用、目前的浏览器支持情况以及兼容解决方案等内容。

只要你略微了解一点HTML、HTTP和JavaScript，就可以顺利阅读本书。

◆ 著　　　　[美] Darren Cook
译　　　　刘 帅
责任编辑　李松峰
执行编辑　李 静　许林玉
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：12.5
字数：258千字　　　　2014年11月第1版
印数：1－3 500册　　　　2014年11月北京第1次印刷
著作权合同登记号　图字：01-2014-5613号

定价：49.00元

**读者服务热线：(010)51095186转600　印装质量热线：(010)81055316**
**反盗版热线：(010)81055315**
**广告经营许可证：京崇工商广字第0021号**

# 版权声明

# O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

# 目录

# 前言

人们对现代网络要求甚高：不仅要求页面美观，加载迅速，还必须有最新、有趣而且有价值的好内容。本书探讨的是一项有助于满足后两个要求的技术：确保你的网站或网络应用用户获取最新的内容，并且毫不妥协地最大限度降低延迟。

本书还关注现实生活中有实用价值的应用。第 2 章以一个非常好玩的范例为基础，第 6 章和第 7 章用的是入门级的例子，其他章节则都是围绕现实生活中无处不在且又无法回避的完整应用。

## 读者对象

在现实生活中，你必须是一个健壮而谦恭、热情而客观的人，你还必须尊老爱幼并且极度热爱互联网。不过，本书没有现实世界那么苛刻，但你需要了解 HTML（超文本标记语言）和 HTTP（超文本传送协议），并且也知道 HTML、CSS（层叠样式表）和 JavaScript 之间的区别。你至少应该能够阅读并理解基本的 JavaScript，以便能理解客户端的代码。（当用到更为复杂的 JavaScript 时，我们会在附注或附录中加以解释。）

本书尽可能保持服务端语言中立，所用到的代码大部分是简单的 PHP 代码，因为对这类应用来说，PHP 言简意赅。只要你了解任何类 C 语言，应该就不难读懂。如果有不懂的地方，请参阅附录 C，那里会介绍一些 PHP 相关知识。第 2 章也会介绍使用 Node.js 的例子。在后面的章节中，如果示例代码是 PHP 专用的，我也会介绍如何用其他语言来实现。

最后，你需要有一台装有网络服务器（如 Apache）的开发机，以便跟着文中的范例学。许多 Linux 系统已经安装了 Apache，如果没有，安装起来也不难。举个例子，在 Ubuntu 系统中，在命令行终端输入 `sudo apt-get install lamp-server`[1] 就能一步安装 Apache、PHP

注 1：如果这个命令不生效，试一下 `sudo apt-get install lamp-server^`。——译者注

和 MySQL。在 Windows 系统中，有一个类似的集成包——XAMPP，它会给你提供所需要的一切。它还有 Mac 版。

## 本书结构

SSE 的核心要素并不复杂，第 2 章仅用了几页的篇幅就介绍了一个完整可运行的范例（包括前端和后端）。在那之前，第 1 章会介绍 HTML5 的一些背景知识、数据推送、可能会用到 SSE 的应用，以及用作替代方案的技术。

从第 3 章到第 7 章，我们创建了一个完整的应用，尽可能使它贴近现实，同时又不会让你为那些不重要的细节而烦恼。这个应用所涉及的领域是金融数据。第 3 章介绍这个应用的核心；第 4 章对它做了一些重构和扩展；第 5 章处理了数据推送应用中会出现的一些棘手的细节，比如复杂的数据、数据源无响应、套接字终止等；第 6 章介绍了一种方案（长轮询），使我们的应用能够兼容那些尚未支持 SSE 的台式机浏览器和手机浏览器；第 7 章展示了两种更优但并不是所有浏览器都支持的方案。第 3 章还用了一些篇幅，介绍如何开发我们的样本应用程序可以推送的真实且可重复使用的数据。虽然这与 SSE 不直接相关，但这非常好地演示了数据推送应用中的易测性设计。

第 8 章涵盖了 SSE 协议的一些要素，我们没有将它们用于在其他章节创建的实用应用程序中。当然，我们介绍了没有使用它们的原因。这就引出了第 9 章来介绍之前章节提到过但未详细阐述的安全方面的内容（cookie、权限控制、跨域）。

## 排版约定

以下是本书中使用的排版约定：

- 楷体
  标示新术语。

- 等宽字体
  用于程序段，或标示文中提到的程序元素，比如变量、函数名称、数据库、数据类型、环境变量、语句以及关键字。

- 等宽粗体
  标示命令行语句或其他需要用户照字面输入的文本。

- 等宽斜体
  标示应该用用户提供的值或由上下文决定的值来替换的文本。



表示提示或建议。



表示一般注解。



表示警告或提醒。

## 使用代码示例

本书使用或提到的源码文件可以从 https://github.com/DarrenCook/ssebook 下载。

本书的主旨是希望能帮你完成你的工作。一般来说，本书所提及的示例代码都可用在你的程序和文档中。除非你要复制书中的大部分代码，否则你无需与我们联系获取我们的许可。比如，在你的程序中使用几个本书中的代码块不需要获取许可。销售或分发含有 O'Reilly 出版的书籍中的代码的 CD-ROM 需要获得许可。引用本书以及本书中的代码来解答疑问不需要获得许可，但在你的产品文档中大量包含本书中的示例代码需要获得许可。

我们欢迎引用本书内容时加以说明，但对这一点并不强求。引用说明通常包括书名、作者、出版社和国际标准图书编号，例如："*Data Push Apps with HTML5 SSE* by Darren Cook (O'Reilly). Copyright 2014 Darren Cook, 978-1-449-37193-7"。

如果你觉得你对示例代码的使用超出了合理使用或上述许可的范围，请发邮件至 permissions@oreilly.com 与我们联系。

## Safari® Books Online




Safari Books Online（http://www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

# 联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreil.ly/data_push_apps_html5-sse。

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

# 第1章

# SSE及其他

SSE（Server-Sent Event，服务端推送事件）是一种允许服务端向客户端推送新数据的HTML5技术。与由客户端每隔几秒从服务端轮询拉取新数据相比，这是一种更优的解决方案。在写作本书时，65%的桌面和移动浏览器原生支持这项技术，但是，本书将介绍如何开发支持超过99%桌面浏览器和移动浏览器的向后兼容解决方案。顺便说一下，10年前我只能用Flash来实现这种数据推送；事情进展太迅速了，本书中已不会有任何用Flash实现的方案。



本书中提到的浏览器占比数据来自超赞的“Can I Use...”网站，网址是http://caniuse.com/eventsource。而该网站的数据则来自StatCounter GlobalStats，网址是gs.statcounter.com/。给那些爱较真的人说明一下，“超过99%”的意思是“在我能接触到的所有桌面或移动浏览器上都可以运行”。所以，如果你发现浏览器支持率没有达到精确的99%，还请见谅。

对那些禁用JavaScript的浏览器来说，不论是SSE还是我们聪明的向后兼容解决方案都不管用。但是，被告知“不可能”是一件让人不爽的事，所以我会介绍一种让这些浏览器也能动态更新的方案（见6.6节）。

接下来，本章会介绍什么是HTML5和数据推送，讨论一些可能会用到SSE技术的应用，并会花点时间对比一下SSE和WebSocket，以及它们和根本不使用数据推送的方案的区别。如果你已经对数据推送有大致的了解，想直接跳到第2章去看代码示例，然后再回到这里，我也可以理解。

## 1.1 HTML5

前面提到 SSE 是一种 HTML5 技术。在现代网络中，HTML 用以指定网页或应用的结构和内容，CSS 用以描述其外观，JavaScript 用以使它具有动态性和交互性。



JavaScript 表达行为，CSS 表达外观；注意，HTML 既表达结构，即逻辑结构（DOM），又表达内容，即数据本身。通常需要更新数据时，并不需要更新结构。正是这种不改变组织结构仅改变数据的诉求，推动了数据拉取和数据推送技术的产生。

大概在 1990 年，蒂姆・伯纳斯 – 李（Tim Berners-Lee）发明了 HTML。官方从未正式发布 HTML 1.0 标准，但在 1995 年末发布了 HTML 2.0。那时候，人们是以月来讨论互联网时代的，因为这项技术发展得太快了。HTML 2.0 因增加了表格、图片上传以及图片映射而得到了增强。1997 年 1 月，以 HTML 1.0 和 HTML 2.0 为基础的 HTML 3.2 发布。同年 12 月，HTML 4.0 发布。当然，中间有过一些微调，那就是 XHTML 出现了，不过它基本上就是今天你在用的 HTML，除非你在用 HTML5。

大部分 HTML5 新增的特性都是可选的，这意味着绝大部分情况下，你可用你所知道的 HTML 4，然后选择一些你想要的 HTML5 特性。HTML5 新增了一些元素（包括直接支持视频、音频，以及矢量和位图绘图）和表单控件，移除了一些在 HTML 4 中不建议使用的东西。但对我们来说，更有意义的是，HTML5 新增了一大堆 JavaScript 应用程序编程接口（API），SSE 就是其中之一。想要了解更多关于 HTML5 的知识，参见维基百科条目[1]。

HTML5 新增特性的正交性意味着，虽然本书的所有代码都是 HTML5 的（如代码第一行 `<!doctype html>` 所示），但除了与 SSE 直接相关的部分，其他的都是你所习惯的 HTML 4，没有用到任何 HTML5 的新标签。

## 1.2 数据推送

SSE 是一种允许服务端向客户端推送新数据（通常称作数据推送）的 HTML5 技术。那么，究竟什么是数据推送？它与我们可能用过的其他技术有什么不同呢？让我先来回答什么不是数据推送。数据推送有两种替代方案：无更新方案和数据拉取方案。

无更新方案（图 1-1）是最简单的。这几乎是所有网络内容的运作方式。

注 1：http://en.wikipedia.org/wiki/HTML5。——译者注

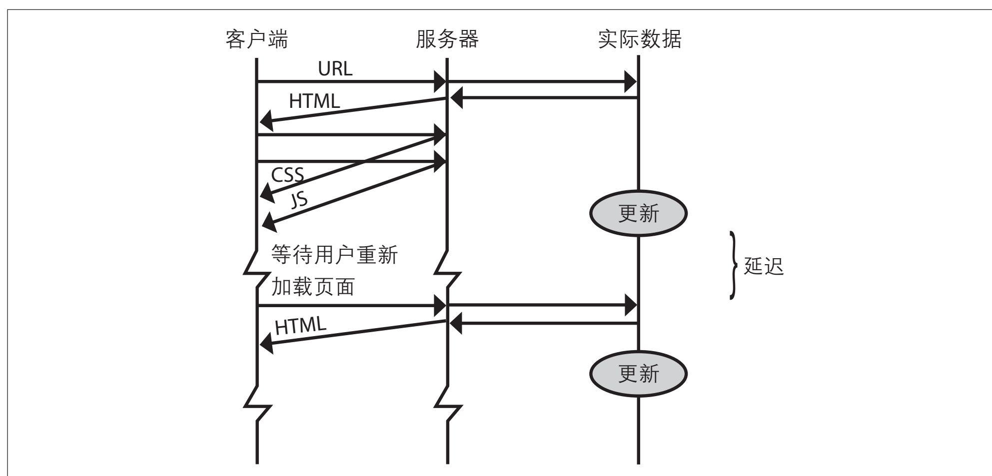



图 1-1：替代方案之一：无更新方案

在浏览器中输入一个 URL，然后你就会得到一个 HTML 页面。之后浏览器会请求图片、CSS 文件、JavaScript 文件等。它们每一个都是浏览器可以缓存的静态文件。如果你正使用的是后端语言，比如 PHP、Ruby、Python 或其他许多为用户动态生成 HTML 的语言，就浏览器而言，它接收到的 HTML 文件与手写的静态 HTML 文件没什么区别。（是的，我知道你会说你可以命令浏览器不缓存内容，但这不是重点，它还是静态的。）

另一种方案是数据拉取（如图 1-2 所示）。

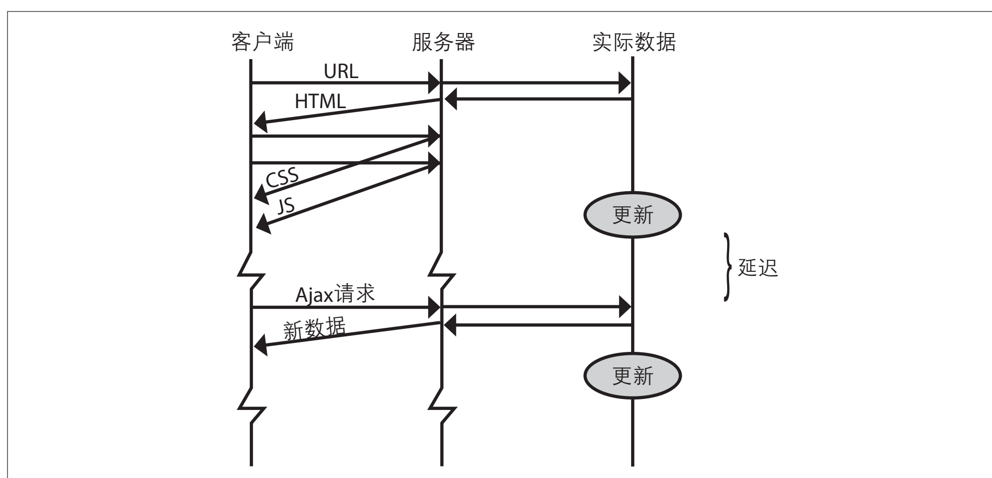



图 1-2：替代方案之二：数据拉取

浏览器会基于一些用户行为，或在一定时间之后，或基于某种别的触发方式，向服务端请求部分或全部最新数据。使用这种简单粗暴的方式，通过 JavaScript 或者一个 meta 标签

（见 6.6 节）都能命令整个页面重新加载。要做到这一点，页面要么是由服务端语言自动生成，要么是定期更新的静态 HTML 页面。

在更复杂的情况下，Ajax 技术只被用于请求最新数据，当收到数据时，JavaScript 函数会利用它来局部更新 DOM。这里有一个很重要的概念：仅请求最新数据，而不是整个 HTML 页面结构。这正是我们所说的数据拉取的要义：仅拉取新数据，并且只更新页面中受到影响的部分。



**术语提示 : Ajax? DOM?**

第 6 章会介绍 Ajax，在没有原生支持 SSE 的浏览器中将会用到它。我不会告诉你它代表什么，那只会让你困惑。毕竟，它不是必须异步，也不是必须使用 XML。不过，Ajax 中的 J 很难讨论，你肯定需要 JavaScript。

DOM 又是什么呢？它是 Domcument Object Model（文档对象模型）的缩略形式。它是代表当前网页的数据结构。如果你用原生 JavaScript 写过 `document.getElementById('x')...`，或者用 jQuery 写过 `$('#x')...`，那么你一直都在使用 DOM。

这些都不是数据推送。数据推送不是静态文件，也不涉及浏览器为最新数据而发起请求。数据推送是由服务端选择向客户端发送新数据（见图 1-3）。

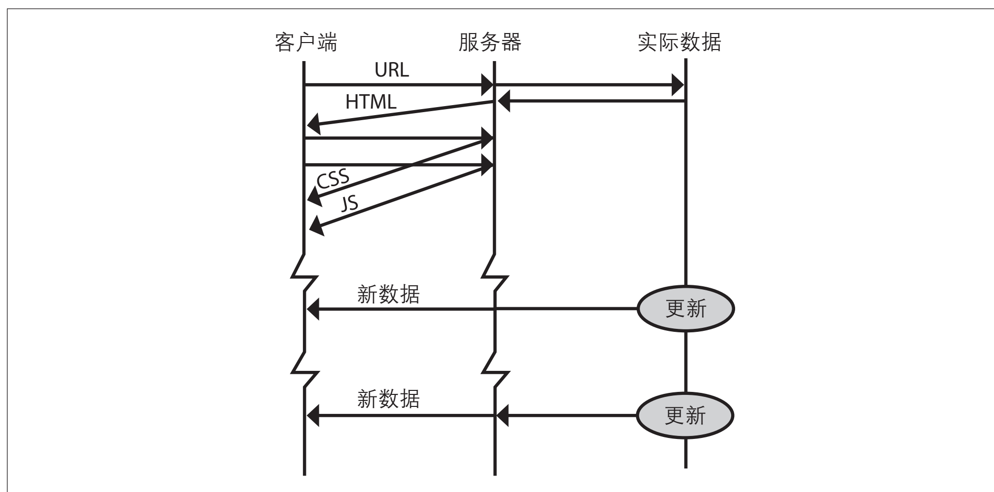



图 1-3：数据推送

当数据源有新数据时，服务端能立刻将它发送给一个或多个客户端，而不用等客户端来请求。这些新数据可以是突发新闻、最新的股价、来自线上朋友的聊天信息、新的天气预报、策略游戏中的下一步等。

数据拉取和数据推送的功能目标是一样的：让用户看到最新数据。但数据推送有一些优势。或许最大的优势就是更低的延迟。假设一个数据包在服务端与客户端之间的传输时间是 100 毫秒，数据拉取客户端以 10 秒为间隔轮询拉取。用数据推送方式，客户端会在服务端读入数据 100 毫秒后看到数据；用数据拉取方式，客户端会在服务端读入数据 100 至 10 100 毫秒（平均 5100 毫秒）后看到数据：这都取决于轮询请求的时间选择。平均来看，数据拉取方式的延迟是数据推送方式的 51 倍多。如果数据拉取方式改为每 2 秒轮询一次，平均时间会下降到 1100 毫秒，仅仅是数据推送方式的 11 倍多。但是，如果没有新数据到达，这也会导致更多的请求和资源（比如带宽、CPU 等）浪费。

在数据拉取方式中，权衡会让你很纠结：要缩短延迟就要提高轮询频次；要节省带宽和连接就要降低轮询频次。延迟和带宽，哪个更重要？当你说"都重要"，那就是你需要数据推送技术的时候了。

## 1.3 数据推送的其他名称

对数据推送的需求可以追溯到 Web 诞生的时候[2]。多年来，人们找到了很多新奇的解决方案，其中大部分都存在人们不期望看到的折中方式。你也许听说过一些其他的技术：Comet、Ajax Push、Reverse Ajax、HTTP Streaming，还一直在想它们之间有什么不同。实际上，这些都属于我们将在第 6 章和第 7 章中探讨的向后兼容解决方案。后来又增加了 SSE，它是一种兼具易用性和高效性的新增 HTML5 技术。如果你的浏览器支持 SSE，它总是[3]比 Comet 技术优越。（本章后面会讨论 SSE 和 WebSocket 的区别。）

顺便说一下，有时你会看到 SSE 被人称为 `EventSource`，因为那是它在 JavaScript 中相关对象的名字。本书会使用 SSE 这个名字，只会在涉及 JavaScript 对象时使用 `EventSource`。

## 1.4 可能会用到SSE的应用

SSE 对什么有用？当你需要用新数据局部更新网络应用时，SSE 便脱颖而出，它不会要求用户执行任何操作。我们将以一个推送外汇价格的应用为例，探索如何实现数据推送和 SSE。我们的目标是每当经纪人那里的欧元 / 美元（欧元兑美元）汇率变化时，新的价格会出现在浏览器上，尽可能像现实中一样及时。

这个例子完全适用于 SSE 处理资料传送的标准：更新频繁、低延迟，并且数据都是从服务端到客户端（客户端不需要将价格数据推送回服务器）。我们示例中的后端会使用杜撰的

注 2：如果你认为数据推送和数据拉取只在 Ajax（在 2005 年开始流行）出现之后才成为可能，再想想，Flash 6 在 2002 年 3 月发布，提供了可以实现数据推送和数据拉取的 Flash Remote 技术，并且还不用因浏览器不同而烦恼（那时几乎每个人的浏览器上都装了 Flash）。

注 3：好吧，并不是总是，参见 1.6 节和 6.3 节附注栏"长轮询是否总是比常规轮询好"。

价格数据，但使用真实数据（无论是汇率还是其他数据）的时候其实都一样。

稍微有点想象力，你就能明白这个例子如何应用到其他领域：在拍卖网络应用中推送最新出价，在售书网站推送新评论，在在线游戏中推送新高分，推送你感兴趣的最新微博或用你感兴趣的关键词推送新闻类文章，推送你在自家后院建立的那个由 Kickstarter[4] 资助的核聚变反应堆中心的最新温度。

还有一些应用会发一些提示，比如像 Facebook 这样的社交网站，当有新消息到来时，应用的某个位置会出现一个浮层，然后渐渐隐去；或者像 Gmail 这样的邮箱服务界面，每当有新邮件时会在你的收件箱里现新的提示；或者连上日历，在会议即将开始前给你发送一条通知；或者在你的某个服务器上的磁盘使用率增高时向你提出警示……尽情畅想吧！

聊天类应用呢？聊天由两部分组成：在聊天室中接收其他人的信息（也可以是其他动态，比如成员进出聊天室、资料修改等）；发送你自己的信息。这种双向沟通一般非常适合用 WebSocket（稍后我们会具体看一下），但这并不意味着它不适合用 SSE。发送自己信息的部分，用古老的 Ajax 请求的方式就挺好。

作为适合用 SSE 的聊天类应用范例，它可用来推送你感兴趣的微博，与此同时，用一个独立的连接供你撰写自己的微博。或者想象一个在线游戏：用 SSE 将新分数发送给所有玩家，而你只需设法在游戏结束时把每个玩家的最终分数发到服务器。或者想象一个多人的实时策略游戏：当前面板位置不断更新，并且通过 SSE 分发给所有玩家，然后在你需要将某个玩家的动作发送到中央服务器时使用 Ajax 通道。

## 1.5 和WebSocket的对比

你可能听说过另一种叫做 WebSocket 的 HTML5 技术，它也能从服务端向客户端推送数据。那如何决定你是用 SSE 还是 WebSocket 呢？概括来说，WebSocket 能做的，SSE 也能做，反之亦然，但在完成某些任务方面，它们各有千秋。

WebSocket 是一种更为复杂的服务端实现技术，但它是真正的双向传输技术，既能从服务端向客户端推送数据，也能从客户端向服务端推送数据。

WebSocket 和 SSE 的浏览器支持率差不多，大多数主流桌面浏览器两者都支持[5]。在 Android 4.3 以及更早的版本中，系统默认浏览器两者都不支持，Firefox 和 Chrome 则完全支持；Android 4.4 中，系统默认浏览器两者都支持；Safari 从 5.0 开始支持 SSE（iOS 系统从 4.0 开始），但直到 6.0 才正确地支持 WebSocket（6.0 之前的 Safari 所实现的 WebSocket 协议存在安全问题，所以一些主流浏览器已经禁用了基于这个协议的实现）。

---

注 4：Kickstarter 是一个创意方案的众筹网站平台。——译者注

注 5：IE 是个例外，即便 IE11 都还不支持原生 SSE，IE10 添加了 WebSocket 支持。

与 WebSocket 相比，SSE 有一些显著的优势。我认为它最大的优势就是便利：不需要添加任何新组件，用任何你习惯的后端语言和框架就能继续使用。你不用为新建虚拟机、弄一个新的 IP 或新的端口号而劳神，就像在现有网站中新增一个页面那样简单。我喜欢把这称为既存基础设施优势。

SSE 的第二个优势是服务端的简洁。我们将在第 2 章中看到，服务端代码只需几行。相对而言，WebSocket 则很复杂，不借助辅助类库基本搞不定（我试过，令人痛苦）。

因为 SSE 能在现有的 HTTP/HTTPS 协议上运作，所以它能直接运行于现有的代理服务器和认证技术。而对 WebSocket 而言，代理服务器需要做一些开发（或其他工作）才能支持，在写这本书时，很多服务器还没有（虽然这种状况会改善）。SSE 还有一个优势：它是一种文本协议，脚本调试非常容易。事实上，在本书中，我们会在开发和测试时用 curl，甚至直接在命令行中运行后端脚本。

不过，这就引出了 WebSocket 相较 SSE 的一个潜在优势：WebSocket 是二进制协议，而 SSE 是文本协议（通常使用 UTF-8 编码）。当然，我们可以通过 SSE 连接传输二进制数据：在 SSE 中，只有两个具有特殊意义的字符，它们是 CR 和 LF，而对它们进行转码并不难。但用 SSE 传输二进制数据时数据会变大，如果需要从服务端到客户端传输大量的二进制数据，最好还是用 WebSocket。

### 二进制数据和二进制文件

如果你正打算通过 WebSocket 或 SSE 传输二进制**文件**，停下来想想是否真的需要这么做。用 HTTP 不是更好吗？免得重复造轮子（权限控制、加密、代理、缓存、长连接，等等）。如果你关心的是连接套接字的有效使用，好好看看 HTTP/2.0[6]。

我说“大量的二进制数据”意思是你需要在浏览器中实现一个二进制网络协议，比如 SSH。如果只是想向用户推送一个新的横幅广告图片，最好的方式是通过 SSE（或者 WebSocket）推送图片的 URL，然后让浏览器通过 HTTP 获取图片。

WebSocket 相较 SSE 最大的优势在于它是双向交流的，这意味向服务端发送数据就像从服务端接收数据一样简单。用 SSE 时，一般通过一个独立的 Ajax 请求从客户端向服务端传送数据。相对于 WebSocket，这样使用 Ajax 会增加开销，但也就多一点点而已[7]。如此一来，问题就变成了“什么时候需要关心这个差异？”如果需要以 1 次 / 秒或者更快的频率向服务端传输数据，那应该用 WebSocket。0.2 次 / 秒到 1 次 / 秒的频率是一个灰色地带，

注 6：参见 http://en.wikipedia.org/wiki/HTTP_2.0，或者 Ilya Grigorik 写的《Web 性能权威指南》（人民邮电出版社）。

注 7：在 HTTP/1.1 中大概几百字节，如果请求中有大量的 cookie 或其他东西，这个量会更多一些。在 HTTP/2.0 中会少很多。

用 WebSocket 和用 SSE 差别不大；但如果你期望重负载，那就有必要确定基准点。频率低于 0.2 次 / 秒左右时，两者差别不大。

从服务端向客户端传输数据的性能如何？如果是文本数据而非二进制数据（如前文所提到的），SSE 和 WebSocket 没什么区别。它们都用 TCP/IP 套接字，都是轻量级协议。延迟、带宽、服务器负载等都没有区别，除非……呃？除非什么？

当你在享用 SSE 的*既存基础设施优势*，并在客户端和服务端脚本之间设了一个网络服务器，区别就显现出来了。一个 SSE 连接不仅使用一个套接字，还会占用一个 Apache 线程或进程，如果用 PHP，它会为这个连接专门创建一个 PHP 新实例。Apache 和 PHP 会使用大量的内存，这会限制服务器所能支持的并行连接数。所以，要做到用 SSE 在数据传输性能上和 WebSocket 完全一样，需要写一个你自己的后端服务器，当然，那些在任何情况下都会用自己的服务器并使用 Node.js 的人，会觉得这有什么稀奇的。第 2 章会介绍怎么用 Node.js 来做这些。

说一下 WebSocket 在旧版本浏览器上的兼容。写这本书的时候，大约超过 2/3 的浏览器支持这些新技术，移动端浏览器的支持率会低一些。依惯例，每当需要双向套接字时，就会用到 Flash，并且 WebSocket 的向后兼容通常是用 Flash 来做，这已经相当复杂了，如果浏览器上没有 Flash，情况更糟。概括来说，WebSocket 难兼容，SSE 易兼容。

## 1.6　什么时候数据推送是错误的选择

这一节要讲的大部分内容，对 HTML5 数据推送技术（SSE 和 WebSocket）和将在第 6 章、第 7 章讲到的向后兼容解决方案都适用，它们的共同点在于，都会为每一个客户端连接打开一个专门的套接字。

首先来考虑静态的情况，不引入数据推送。每当用户打开一个页面，在浏览器和服务器之间就会打开一个套接字连接。服务器收集信息然后返回给用户，可能会很简单，就像从磁盘上加载一个静态 HTML 文件或一张图片一样；也可能会很复杂，就像要运行一段用以连接很多数据库的后端语言，将 CoffeeScript 编译成 JavaScript，然后把它们结合到一起（用一个服务端模板）并返回。这里的关键点是，一旦返回了所需的信息，套接字就会关闭[8]。每个 HTTP 请求都会打开一个这种生命期相对较短的套接字连接，但这些套接字是服务器上的有限资源，每当它们完成既定任务，就会被回收以循环再利用。这真是相当环保啊，居然没有政府部门给它捐款。

现在对比看一下数据推送。一个请求永远不会完成：总是有更多信息要发送，所以套接字会

注 8：事实上大部分请求使用 HTTP 持久连接，它会共享第一个 HTTP 请求和图片之间的套接字，这个连接会在停止活跃几秒（Apache 2.2 中是 5 秒）后关闭。提到这个只是为了解释清楚，并不影响普通网络方案和数据推送方案的比较。

一直保持打开状态。显然，因为它们是有限的资源[9]，所以同一时刻的 SSE 连接数会有限制。

可以这样想象一下：你在为你最新的一个应用提供电话服务支持，有 10 个接线中心员工为 1000 个用户提供服务。当一个用户遇到问题并拨打客服电话，其中一个接线员接线，帮他解决问题，然后挂线。闲的时候，一些接线员没有电话可接；忙的时候，全部 10 个接线员都在忙而且还有新的客户呼叫在排队，直到其中有一个接线员挂线。这就是典型的网络服务模式。

但是，现在想象一下你有一个客户打进来并且说："我现在没有问题，但我在接下来的几个小时会用你们的软件，并且如果遇到问题，我希望能立即答复，并且不会有被搁置的风险，所以，请问您可以就这样保持电话畅通吗？"如果你提供这项服务，而这位客户没有问题要问，那么在与他保持通话的那几个小时，呼叫中心 10% 的服务资源就浪费了。如果 10 个客户这样做，其他 990 位客户就无法呼叫了。这就是数据推送模式。

但这并不总是坏事，想象一下，如果那个用户一下午每隔几秒钟就有一个问题。这种情况下保持电话畅通不但没有浪费 10% 的服务资源，反而会增加。如果他每个问题都要重新打一个电话（就像数据拉取），想一下接线员花在接线、验证客户身份、调出他账户的时间，甚至还有在通话结束时礼貌性地说再见的时间。如果他每次呼叫都是由不同的接线员接线，他们还要花点时间聊一下问题背景之类的，这也会降低服务效率。而保持电话畅通不仅使你的客户更满意，也会提高你呼叫中心的工作效率。这是数据推送模式最适合的场景。

前面提到的外汇价格的例子就很适合用 SSE 来做，有大量的价格变化，并且低延迟很重要：用户只能以当前的价格交易，而不是 60 秒以前的。另一方面，考虑一下大范围的天气预报，气象局会每半小时发布一次最新的天气预报，但多数时候天气不是从"晴"变成别的，并且延迟也不是很重要的问题。如果天气预报员播报的天气预报，不是从"晴"变成"多云"，那这则天气预报真的有意义吗？这是否值得保持一个套接字一直打开，或者每 30 分钟或 60 分钟直接从气象服务器拉取（数据拉取）就足够了？

那有什么是发生频率不高但又需要关心延迟问题的事件呢？假如政府将在上午 8:30 发布关于经济增长的公告，我们希望在发布时立刻将公告更新到我们的网络应用上，要怎么做？这种情况，最好是设置一个计时器，在公告即将发布之前调用一个 Ajax 长轮询（见第 6 章）。提前几个小时或者几天保持一个套接字一直打开是一种浪费。

一种类似的情况是可预见的停工时刻，回到接收实时外汇价格的例子，没有必要在周末还保持连接，应该在周五下午 5 点（纽约当地时间）关闭连接，并且设置一个定时器在周日

注 9：多少限制呢？这取决于服务器操作系统，可能是每个 IP 地址 60 000 个。防火墙或负载均衡器也可能会做一些限制。服务器的内存也是个关键因素。这个问题很不好说，所以我的建议是在实际使用的操作系统中找出这个限制。

下午 5 点重新打开。如果服务器是建立在一个运行即收费的云服务上的，那这意味着可以在周五晚上关闭一些服务请求，这可以节省大概 28% 的开支！参阅 5.4 节 。

## 1.7　决策、决策还是决策

前面两节从正反两方面探讨了数据拉取、SSE 和 WebSocket，但怎么知道哪个更适用？这个问题很复杂，它是以应用的表现、用户对延迟的预期相关的商业决策、主机费用方面的商业决策以及用户和你的开发人员使用的技术为基础的。这里有一些你需要自行思考的问题。

- 服务端事件发生得有多频繁？
  频率越高越适合用数据推送（不论 SSE 还是 WebSocket）。

- 客户端事件发生得有多频繁？
  如果事件触发的频率低于 0.2 次 / 秒，尤其是低于 1 次 / 秒，用 WebSocket 比用 SSE 好。如果频率低于 0.1 次 / 秒到 0.2 次 / 秒左右，那用哪个都可以。

- 服务端事件是不是不但发生频率不高而且还发生在可预见的时刻？
  当这些事件的触发频率低于 1 次 / 分钟，用数据拉取更好，因为它不需要保持一个套接口一直打开。需要注意大量客户端同时试图连接服务器的情况。

- 延迟问题有多关键？给个量化的数据。
  半秒延迟是否会让用户烦躁？ 60 秒的延迟是否也不是什么问题？
  用户越介意延迟，数据推送比数据拉取就越有优势。

- 是否需要从服务端向客户端推送二进制数据？
  如果有大量的二进制数据，用 WebSocket 比用 SSE 更好（在这方面 XHR 轮询也比 SSE 更好）。
  如果是少量的二进制数据，可以对它进行编码，然后用 SSE，区别（相对 WebSocket）是会多几百字节。

- 是否需要从客户端向服务端推送二进制数据？
  用 `XMLHttpRequest`[10]（比如 Ajax，这是 SSE 从客户端向服务端发送信息的方式）和用 WebSocket 处理二进制数据没什么区别。

- 大部分的用户是用有线连接还是移动连接？
  使用 LTE WiFi 路由器或者被限速的笔记本用户，视为移动连接用户；通过很强的 WiFi 连接到光纤上游连接的手机用户，视为有线连接用户。重要的是连接，而不是电脑性能或屏幕尺寸。

---

注 10：严格来说，`XMLHttpRequest` 的第 2 版，参见 http://caniuse.com/xhr2，IE9 以及更早的版本和 Android 2.x 都还没支持，但那些支持 WebSocket 和 SSE 的浏览器也没有，所以也不影响决策。

要知道，移动连接会有更多的延迟，尤其是当连接需要唤醒的时候。这使得在移动连接上使用数据拉取（轮询）方案比在有线连接上要糟糕。

同样，一个超载的 WiFi 连接（比如在一个有很多人的咖啡馆）会丢失越来越多的数据包，其表现更像一个移动连接，而不是有线连接。

- 耗电量是否是移动用户关注的重点？
  需要在延迟和耗电量之间做出权衡。数据拉取（除非你知道数据什么时候出现，从而预知什么时候开始轮询）通常比数据推送（SSE 或 WebSocket）要糟糕。

- 推送的数据是否相对来说比较小？
  一些 3G 移动连接有一个专门的低电量模式，可以用来传输小数据（200 bit/s 到 1000 bit/s）。但那不是重点，更重要的是一个大的消息会被分割成 TCP/IP 片段发送，有一个片段丢失，就要重发。TCP 会确保按发送数据的顺序接收数据，所以这个丢失的数据包会阻碍整个数据的处理，同时也会阻塞后面数据的接收。所以，在不稳定的连接中（比如，移动连接，超载的 WiFi 连接），发送的数据越大，需要发送的额外数据包就越多。
  考虑一下用数据推送作为控制通道，告诉浏览器直接请求大文件，浏览器很有可能用自己的套接字处理，因此不会阻塞数据推送套接口（这之所以存在，是因为你认为延迟问题很重要）。

- 数据推送是 Web 应用的次要特性还是主要特性？是否有开发者资源的短缺？
  SSE 用起来简单，而且是简洁地运行在现有的基础设施上，比如 Apache。这就节省了测试时间。项目越大，你拥有的开发人员越多，这个问题就越不重要。



想了解更多关于前面几节中讨论的技术细节，尤其是当效率和处理高负载是你首要关心的问题，我强烈推荐你看看 Ilya Grigorik 写的《Web 性能权威指南》（人民邮电出版社）。

## 1.8　带我看代码吧

简而言之，如果想要网站更迅速地刷新数据，并且你现在正用 Ajax 轮询，或者页面重载，或正考虑使用这些方案，或者想用 WebSocket 又觉得它水平太低了，那么 SSE 就是你一直在找的技术。话不多说，赶快到下一章看一个数据推送的 Hello World 吧。

# 第 2 章

# 玩转SSE

本章介绍的是一个基于 SSE 的从服务端向客户端实时推送数据的简单示例，包括前端和后端。本章不会涉及 SSE 的一些不常用特性（这些内容将在第 5 章、第 8 章和第 9 章介绍），也不会针对不支持 SSE 的旧版浏览器做兼容处理（参见第 6 章和第 7 章）。但即便如此，本章的示例也能在大部分主流浏览器的最新版本上运行。



任何最新版的 Firefox、Chrome、Safari、iOS 版 Safari 或 Opera 都没有问题，在 IE11 以及更早的版本上则无法运行，Android 4.3 以及更早版本的系统默认浏览器上也无法运行。要在 Android 手机或平板电脑上测试本章示例，请安装 Android 版 Chrome 或 Firefox。也可以选择长轮询这种向后兼容方案，第 6 章将介绍这种方案。想知道哪些浏览器原生支持 SSE，请参见 http://caniuse.com/eventsource。

如果你已经迫不及待地想要试一下，就把 basic_sse.html 和 basic_sse.php 放到 Apache（或者你所使用的其他网络服务器软件）的同一目录下[1]，服务器可以是本地的，也可以是远程的。如果文件放在本地服务器一个叫 SSE 的目录下，那么打开 http://localhost/sse/basic_sse.html 会看到页面上每秒出现一个时间戳，并且很快就会填满整个页面。

## 2.1　最简单的示例：前端

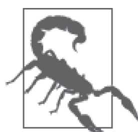
我会慢慢讲这个例子，说不定你需要复习一些 HTML5 或 JavaScript 方面的知识。首先，

注 1：此时此刻，务必将你的 HTML 和服务端脚本放在同一个服务器中，第 9 章会介绍跨域解决方案，使前端页面（在某些浏览器上）可以访问不同服务器上的脚本。

我们来创建一个最简单的 HTML 文件，用 HTML/head/body 标签搭个架子。第一行是 HTML5 的文档类型声明，它比你在 HTML 4 中可能看到的文档类型声明要简单很多。在 <head> 标签中指定编码格式为 UTF-8，但这不是因为示例中用了什么外星字符，而是因为如果不指定，一些校验工具会抱怨。

```
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Basic SSE Example</title>
    </head>
    <body>
        <pre id="x">Initializing...</pre>
    </body>
</html>
```

这里用了一个 `<pre>` 标签，`id` 是 `"x"`。之所以用 `<pre>` 而不是 `<p>` 或者 `<div>`，是为了确保（包含换行的）数据能以它被接收时的格式呈现，而不会被修改或格式化。



使用服务端数据之前最好做一下检查，以防潜在的 JavaScript 注入攻击。

初始状态下，`<pre>` 块内写死了内容“Initializing...”，我们会用数据替换那段文本。

### jQuery 和 JavaScript

如果你一直在用 jQuery，用 `$("#x")` 获取 HTML 中 x 的对象引用的等效方法是 `document.getElement-ById("x")`。要替换这段文本，我们将其赋值给 `innerHTML`。要在现有文本上追加，要用 `+=` 而不是 `=`，如下所示：

```
// 与 $("#x").html("New content\n")；等效的原生 JavaScript 代码
document.getElementById("x"). innerHTML = "New content\n"
// 与 $("#x").append("Append me\n")；等效的原生 JavaScript 代码
document.getElementById("x").innerHTML += "Append me\n"
```

接下来在 HTML 主体底部添加一个 `<script>` 块：

```
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Basic SSE Example</title>
    </head>
    <body>
        <pre id="x">Initializing...</pre>
```

```
        <script>
        var es=new EventSource("basic_sse.php");
        </script>
    </body>
</html>
```

我们创建了一个 EventSource 对象，将要连接的 URL 作为它唯一的参数，这里要连接到 basic_sse.php。恭喜，我们现在已经有了一个可以运行的 SSE 脚本。仅此一行代码就会连接到后端服务器，然后浏览器就能接收到连续的数据流了。但如果你运行这个示例，会觉得它的确没什么价值，就算你真这么想，我也不怪你。

要看到 SSE 发送的数据，需要处理"信息"事件。SSE 是异步的，这意味着程序不需要停在那儿等着服务器返回，也不需要通过轮询去看是否有新数据。JavaScript 一如既往地存在着，与用户交互，做出笨拙的动画，发送按键给政府机关，以及任何 JavaScript 可以做的事。当服务器有话要说时，就会调用一个指定的函数，这个函数被称为"事件处理程序"，可能你也听过"回调函数"这个称呼。在 JavaScript 中，对象触发事件，并且每个对象都有一组可以被侦听的事件。要绑定一个事件处理程序，可以写一段这样的代码：

```
es.addEventListener('message', FUNCTION, false);
```

以 es. 开头表示要侦听刚刚创建的 EventSource 对象的事件。第一个参数是事件名称，这里是 'message'；第二个参数是处理这个事件的函数 [2]。

用来处理事件的 FUNCTION 函数附带一个参数，是要处理的事件，通常用 e 表示。e 是一个对象，而我们关心的是 e.data，它包含了服务端发送给我们的新消息。可以单独定义这个回调函数，然后把函数名作为第二个参数赋给 addEventListener，但更常用的是一个匿名函数，省得那一行函数弄乱我们的代码（还要想一个合适的函数名）。把这些一起放到示例代码中，如下所示：

```
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Basic SSE Example</title>
    </head>
    <body>
        <pre id="x">Initializing...</pre>
        <script>
        var es = new EventSource("basic_sse.php");
        es.addEventListener("message", function(e){
            // 在这里使用 e.data
            },false);
        </script>
    </body>
</html>
```

注 2：第三个参数 false 表示在冒泡阶段处理事件，而不是捕获阶段；管它呢，用 false 就是了。

它还是什么都没做！所以在回调函数的函数体中，我们让它把 e.data 插入到 <pre> 标签中（每条消息前面加了一个换行符以便每条信息自成一行）。最终代码如下所示：

```
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Basic SSE Example</title>
    </head>
    <body>
        <pre id="x">Initializing...</pre>
        <script>
        var es = new EventSource("basic_sse.php");
        es.addEventListener("message", function(e){
            document.getElementById("x").innerHTML += "\n" + e.data;
            },false);
        </script>
    </body>
</html>
```

终于，我们看到有一行显示“Initializing...”，然后每秒都出现一个新的时间戳（如图 2-1 所示）。

```
初始化中...
2014-01-08   15:35:51
2014-01-08   15:35:52
2014-01-08   15:35:53
2014-01-08   15:35:54
2014-01-08   15:35:55
```

图 2-1：basic_sse.html 运行几秒后的效果

可以为其他 EventSource 事件写回调函数，但这都不是必需的，在后面第一次用到时会作介绍。

## 2.2　使用jQuery吗

现在很多人用 jQuery。但前面提到的 SSE 样例代码太简单了，用 jQuery 也简化不了多少。下面是用 jQuery 重写的极简示例代码，供参考：

```
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Basic SSE Example</title>
        <script src="//code.jquery.com/jquery-1.11.0.min.js"></script>
    </head>
```

```
    <body>
        <pre id="x">Initializing...</pre>
        <script>
        var es = new EventSource("basic_sse.php");
        es.addEventListener("message", function(e){
            $("#x").append("\n" + e.data);
            },false);
        </script>
    </body>
</html>
```

下面这个版本（本书源代码里的 basic_sse_jquery_anim.html）在每次更新时用了淡入淡出动画，使它看上去更漂亮，此外还用替换取代了追加，所以只会显示最新的时间戳。

```
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Basic SSE Example</title>
        <script src="//code.jquery.com/jquery-1.11.0.min.js"></script>
    </head>
    <body>
        <pre id="x">Initializing...</pre>
        <script>
        var es = new EventSource("basic_sse.php");
        es.addEventListener("message", function (e) {
            $("#x").fadeOut("fast", function () {
                $("#x").html(e.data);
                $("#x").fadeIn("slow");
                });
            }, false);
        </script>
    </body>
</html>
```

## 2.3　最简示例：后端

我们学习的第一个后端（服务端）示例是用 PHP 写的，如下所示：

```
<?php
header("Content-Type: text/event-stream");
while (true) {
    echo "data:".date("Y-m-d H:i:s")."\n\n";
    @ob_flush(); @flush();
    sleep(1);
    }
```

这段代码就像前端代码一样，超短，对吧？没有库，没有依赖，只是几行简单的普通 PHP 代码。就像前端一样，这里还可以做更多事情，但也都不是必需的。

代码第一行 `<?php` 表明这是一个 PHP 脚本，然后用 `header()` 函数返回一个 MIME 类型的

text/event-stream，text/event-stream 是专门为 SSE 设计的 MIME 类型。接下来是一个无限循环（while(true){...} 是 PHP 的惯用做法），在这个循环中，每秒输出当前时间戳。

SSE 协议要求消息数据（时间戳）加上 data: 前缀，并在后面加了一个空行。所以假设是从 2014 年 2 月 28 日下午 1 点开始，输出的时间戳如下所示[3]：

```
data:2014-02-28 13:00:00

data:2014-02-28 13:00:01

data:2014-02-28 13:00:02

data:2014-02-28 13:00:03

...
```

@ob_flush;@flush(); 这一行是做什么用的呢？它告诉 PHP（以及 Apache）立即将数据返回给客户端，而不是缓冲起来成批发送。@ 前缀的意思是忽略错误，用在这里很恰当：如果没有数据可以冲掉，ob_flush() 会报错，但这并不是我们需要关心的。（你可能想问，ob_flush() 是否必须放在 flush() 前面？是的，必须！）

## PHP 错误控制

对于 PHP 专家来说，@ 会很慢，但在如上文所述的场景中，调用 2 次运行时间大概多了 0.01 毫秒，如下所示。所以，只要不是把它放在一个紧凑的循环里，大可以放心。@foo() 是一种简写，调用 foo() 之前，它是 $prev=error_reporting(0); 的简写，调用 foo() 之后，它是 error_reporting($prev) 的简写。所以，如果你确实很在意性能，又需要在循环里使用 @foo()，也熟悉这个语句的含义，那最好还是把那些语句（$prev=error_reporting(0);error_reporting($prev)）放在循环外面。

ob_flush 用来阻止程序报 E_NOTICE，所以更好的完整写法是这样的：

```
$prev = error_reporting();
error_reporting($prev & ~E_NOTICE);
...
ob_flush();
flush();
...
error_reporting($prev);
```

http://bit.ly/1gCNyfX 表明 flush() 永远不会报错，所以它前面的 @ 可以删掉，只需保留 ob_flush() 前面的部分。http://bit.ly/1elPD1S 列出了 ob_flush 可能会抛出的警告。

无限循环让你感到紧张了吗？这里还好啦。一个 SSE 连接占用一个 Apache 的线程 / 进程，

注 3：如果输出结果没有空行，尝试将代码中的 "\n\n" 替换成 PHP_EOL.PHP_EOL。——译者注

但只要浏览器关闭连接（不论是通过 JavaScript，还是用户关闭浏览器窗口），这个套接字就会关闭，然后 Apache 会关闭这个 PHP 实例。

你是否会担心缓存问题？不论是客户端的还是中间代理服务端的。的确，缓存对 SSE 来说是非常糟糕的，因为让用户及时获取最新信息才是我们的重点。在我的测试中，客户端从没有做任何缓存。因为这是个极简的例子，所以我有意忽略了缓存问题。为确保万无一失，其他章节的例子里会明确指明禁用缓存（参见 5.10 节）。



在使用 SSE 时还需要注意一件事，即浏览器会终止一个失活的连接。比如，有些版本的 Chrome 浏览器会在连接失活 60 秒后关闭连接（然后重新打开）。在我们的真实应用中，我们会处理这个问题（参见 5.3 节）。不过现在还不需要，因为服务端绝不会失活：我们每秒都会输出数据。

## 2.4　基于Node.js的后端

在这一节中，我们会在后端使用 Node.js。它和你所知道的浏览器中的 JavaScript 是一样的，甚至连一些库也是一样的（字符串、正则表达式、日期等），但它是用在服务端的，还扩展了模块加载。用 Node.js 时最需要注意的是，在默认情况下，一切都是无阻塞的，换句话说，就是异步的：异步编程需要用不同的思路。但正是这种无阻塞、事件驱动的特点使它非常适合用来做数据推送应用。

在前面使用的 PHP 服务器解决方案，其更好的叫法是“Apache+PHP”，因为是用 Apache（或者其他服务器软件）处理 HTTP 请求事务（以及一堆其他事务，比如授权），然后只是用 PHP 处理请求本身的逻辑。且不说这让我们的示例代码相当小，这也是 PHP 的最常见用法。Node.js 自带网络服务库，这是很多人用它做网络内容服务的方式，也是这里将要用到的方式。



不要陷入无休止的语言战争。任何语言，在你熟悉它之前，都会让你觉得不舒服。一旦习惯了，即便有不舒服的地方，你也知道怎么处理。PHP 和 Node.js 真正强大的地方很类似：非常流行，相关开发人员很好找，并且有很多实用的扩展。

### 2.4.1　基于Node.js的最简Web服务器

在介绍如何用 Node.js 支持 SSE 之前，先看一下基于 Node.js 的最简 Web 服务器代码：

```
var http = require("http");

http.createServer(function(request,response) {
```

```
    response.writeHead(200,
        { "Content-Type": "text/plain" }
        );
    var content = "Hello World\n";
    response.end(content);
    }).listen(1234);
```

第一行引入了 http 库，这是 CommonJS 中引入模块的方式。然后可以仅用一行代码启动一个 HTTP 服务器：

```
http.createServer(myRequestHandler).listen(port);
```

这一行代码很强大：它将开始监听指定的端口号，处理所有的 HTTP 协议，处理多个客户端，并且当客户端连接时会调用专门的请求回调函数。在默认情况下，它会监听所有的本地 IP 地址。如果想让它监听 127.0.0.1，具体指定如下：

```
http.createServer(myRequestHandler).listen(port,"127.0.0.1");
```

按照惯例，请求回调函数以匿名函数实现，我们的示例遵循这一惯例。这个函数有两个参数：一个是 `http.ClientRequest`[4] 对象的实例 `request`，另一个是 `http.ServerResponse`[5] 对象的实例 `response`。

`request` 参数告诉服务端客户端在请求什么，`response` 对象返回客户端请求的内容。这个最简示例中完全忽略了用户请求：任何请求获得的东西（`content` 字符串）都是一样的。这里两次调用了 `response` 对象，其中第一次是指定状态（HTTP 状态码 200 的意思是“成功”）和请求头的内容类型（这里是纯文本，不是 HTML），第二次调用是 `response.end(content)`，这其实是两次调用的一种简写：发送数据给客户端（可以选择指定编码）的 `response.write(content)` 和表示全部发送完成的 `response.end()`。

要测试这段代码，需把它保存为 basic_sse_node_server1.js，在命令行中运行 node basic_sse_node_server1.js，然后在浏览器中访问 http://127.0.0.1:1234，你就能看到“Hello World”。

## 2.4.2 用Node.js做推送

上一节我们忽略了用户输入，并输出静态纯文本内容。在接下来的代码块里我们继续忽略用户输入，但是会输出动态文本——当前时间戳，就像之前 PHP 代码所做的：

```
var http = require("http");

http.createServer(function(request,response) {
    response.writeHead(200, { "Content-Type": "text/event-stream" });
    setInterval(function () {
```

注 4：参见 http://nodejs.org/api/http.html#http_class_http_clientrequest。

注 5：参见 http://nodejs.org/api/http.html#http_class_http_serverresponse。

```
        var content = "data:" +
            new Date().toISOString() + "\n\n";
        response.write(content);
        }, 1000);
    }).listen(1234);
```

第一个变化是输出 `text/event-stream` 内容类型，这不是重点。但与前述示例相比最大的变化是加入了 `setInterval(..., 1000)`，用来每秒运行一段代码。在 PHP 中，我们用一个无限循环和一个 sleep(1) 语句达到同样的目的。如果在 Node.js 中也这样做会阻塞整个网络服务器，其他客户端就不能连接了。写一个 Node.js HTTP 服务器时，尽可能快地退出请求回调函数很重要，所以 Node.js 的方式是用 `setInterval`。每秒被调用 1 次的那段代码相当简单。`data:` 前缀和 `\n\n` 后缀是 SSE 协议规范所要求的，`new Date().toISOString()` 是 JavaScript 中获取当前时间戳的常用方式。

在命令行工具中，输入命令 node basic_sse_node_server2.js 来启动服务。现在不要尝试在浏览器上测试（它不会运行的）。如果安装了 curl，可以通过在命令行工具中输入 curl http://127.0.0.1:1234/ 来测试。每秒会出现一个新的时间戳，每条数据之间有一个空行，数据如下：

```
data:2014-02-28T13:00:00.123Z

data:2014-02-28T13:00:01.145Z

data:2014-02-28T13:00:02.140Z

data:2014-02-28T13:00:03.142Z

...
```

### 一些改进

有很多方法可以用来改进这段脚本，虽然这与本章的**最简**主题相悖。在代码最上面添加这一行代码：

```
var port = parseInt( process.argv[2] || 1234 );
```

然后修改一下脚本的最后一行，修改后的代码如下所示：

```
...
}).listen(port);
```

这样就可以在命令行中指定要监听的端口号。如果服务端还没有任何正在运行的网络服务器，可以指定 80 端口来把这段脚本当做 root 来运行。

下一个变化是添加一些调试信息以便观察它是如何运作的，用下面三行代码替换 `response.write(content);`：

```
var b = response.write(content);
if(!b)console.log("Data queued (content=" + content + ")");
else console.log("Flushed! (content=" + content + ")");
```

就像在浏览器中，JavaScript console.log() 可以让程序员观察程序的动静。如果数据被彻底擦除，response.write() 会返回 true。大多数时候都是这样，这是好事。如果数据被缓存到内存，response.write() 会返回 false。这意味着在 response.write() 返回时，数据还没有发送到客户端。如果发送数据的频率太快（这很难看出来。即便把时间间隔从 1000 毫秒改为 1 毫秒也不会被视为“太快”，但把 setInterval 去掉，用 while(true){...} 循环就可以），或者套接字损坏了，就会发生这种情况。

重新启动 node 服务器，然后重新启动 curl 客户端，等到出现一些数据的时候按下 Ctrl-C 来关闭 curl 客户端，去 node 窗口看一下它怎么继续发送数据。啊，噢……这正是用 Apache+PHP 时，Apache 处理的一些其他事情。

我们要做的是让服务端感知到服务端断开连接，这能通过侦听 close 事件实现。close 事件是 request.connection 的一部分，所以可以通过添加下面这段代码来响应这个事件：

```
request.connection.on("close", function(){
    response.end();
    clearInterval(timer);
    console.log("Client closed connection. Aborting.");
    });
```

这段代码需要放在 setInterval 调用**之后**。在那之前，像下面这样获取 setInterval 的返回值：

```
var timer = setInterval(function(){
    ...
```

现在，回调函数会在客户端断开连接时触发，然后把 response 彻底关闭，同时也关闭那个每秒钟蹦跶一次的计时器。

如果你看了本书源码中的 basic_sse_node_server3.js，会发现许多额外的 console.log() 语句。

### 2.4.3　在浏览器中运行

首先，启动 node 服务器（node basic_sse_node_server3.js），找到本章前面提到的 basic_sse.html，用编辑器打开它，然后找到这一行：

```
var es = new EventSource("basic_sse.php");
```

把它改成使用监听 1234 端口的 Node.js 服务器：

```
var es = new EventSource("http://127.0.0.1:1234/");
```

现在在浏览器中打开 basic_sse.html（前提是已经在 80 端口上运行 Apache，至少能运行 HTML 文件）。

什么都没发生，你会看到“Preparing...”，然后就没有下文了。为什么？因为 HTML 文件是从 80 端口加载的，却试图连接到 1234 端口。不同的端口号足以被视为不同的服务器，向一个不同的服务器建立连接是不允许的（因为安全原因）。第 9 章会介绍跨域资源共享（CORS），它给服务器提供了一种方式，来表示它们想要接受从其他服务器加载内容的客户端连接。替代方案是用 Node.js 向客户端传送 HTML 文件，这是 Node.js 的一般做事方式。

（在继续之前，把 basic_sse.html 里的 SSE 连接改回 basic_sse.php。）然后，脚本能读取本地文件系统中的文件，把下面这行代码加到脚本的最上面：

```
var fs = require("fs");
```

那么，最大的变化发生在请求回调函数的最上方。在最上面加上下面这段代码：

```
if (request.url != "/basic_sse.php") {
    fs.readFile("basic_sse.html",
        function (err, file) {
            response.writeHead(200,
                {"Content-Type": "text/html"}
                );
            response.end(file);
        });
    return;
    }
```

当得到一个具体的 URL 时，把它当成对流式传输的请求，接下来（注意 `!=`）把这个 HTML 文件返回。`readFile()` 是 Node.js 的一种异步操作。传入一个文件名，然后当文件加载完成时，由一个异步函数处理它的内容。在等待文件加载完成的同时，请求回调函数返回。当文件加载完成时，只是简单地把它以 text/html 的内容类型返回给客户端，然后用 `end()` 关闭连接。

现在可以在浏览器中访问 http://127.0.0.1:1234 了。

## 修改 HTML 文件

什么情况？为什么在前面的代码片段中提到了 PHP？你已经陷入和 PHP 阵营的语言战争中，陷得太深以至于要在他们的茶里下毒，向老板抱怨他们的个人卫生问题，并且通过邮件给他们发去超过 35 篇文章的链接，让他们看看异步编程多么重要、多么简单。现在看起来你在用 Node.js 处理 PHP 内容。原因很简单：basic_sse.html 原本是为了连接到 PHP 脚本而写的，我不想再多写一个文件。

好吧，这很好解决。在从磁盘加载文件之后，发送给客户端之前，为什么不修改一下要连接的 URL？按下面粗体部分修改：

```
    if (request.url != "/sse") {
        fs.readFile("basic_sse.html",
            function (err, file) {
                response.writeHead(200,
                    {"Content-Type": "text/html"}
                    );
                var s = file.toString();
                s = s.replace("basic_sse.php", "sse");
                response.end(s);
                });
        return;
        }
```

顺便说一下，file 实际上是一个 buffer，而不是一个字符串（因为它可能包含二进制数据），这也是首先要把它转化成字符串的原因。

在本书源代码的 basic_sse_node_server.js 文件中能找到本节用到的代码，以及上面两个附加内容中的代码。下面是它的全部代码：

```
var http = require("http"), fs = require("fs");
var port = parseInt(process.argv[2] || 1234);

http.createServer(function (request, response) {
    console.log("Client connected:" + request.url);
    if (request.url != "/sse") {
        fs.readFile("basic_sse.html", function (err, file) {
            response.writeHead(200, {'Content-Type': 'text/html' });
            var s = file.toString(); //file 是 buffer
            s = s.replace("basic_sse.php", "sse");
            response.end(s);
            });
        return;
        }
    // 下面是处理 SSE 请求，它永不返回。
    response.writeHead(200, { "Content-Type": "text/event-stream" });
    var timer = setInterval(function () {
        var content = "data:" + new Date().toISOString() + "\n\n";
        var b = response.write(content);
        if (!b)console.log("Data got queued in memory (content=" + content + ")");
        else console.log("Flushed! (content=" + content + ")");
        }, 1000);
    request.connection.on("close", function () {
        response.end();
        clearInterval(timer);
        console.log("Client closed connection. Aborting.");
        });
    }).listen(port);
console.log("Server running at http://localhost:" + port);
```

它比 basic_sse.php 的代码要多很多，因为需要处理在 Apache+PHP 解决方案中 Apache 已经处理的事务。

## 2.5 华丽退场

这就是 SSE 的 Hello World 示例。前端和后端都只需几行代码，不能再简单了，对吧？在接下来的 5 章里，我们会在这些知识的基础上，使它更复杂、更健壮，以便适用于几乎所有的桌面和移动浏览器。

# 第3章

# 迷人的真实数据推送应用

本章会在上一章所创建代码的基础上实现一个真实的（全方位地模拟真实场景）数据推送应用（可以在 3.1 节中看到所选的问题领域）。本章和接下来两章的代码仍然只能在支持 SSE 的浏览器上运行。第 6 章和第 7 章会介绍如何修改前端和后端代码来兼容旧版浏览器。



因为这一章只是关于 SSE 的，如果要在 Android 设备上测试，请安装 Android 版的 Chrome 或 Firefox。如果在 Windows 系统上测试，请安装 Firefox、Chrome、Safari 或者 Opera。好吧，你肯定至少已经安装了其中的一个，你说过你是一个专业开发人员！

本章会包含一些可能和应用本身关系不大的 PHP 后端代码，建议你至少浏览一下这些代码，因为后面几章以它为基础，而且它一步一步地展示了一种对数据推送系统进行单元测试和功能测试的方法。

## 3.1　问题领域

本章和接下来的几章涉及金融行业的问题领域。像软件行业一样，它也有一些生涩的行业术语，所以我会介绍一些你可能会遇到的术语，以及足够的背景信息，以帮助你理解应用的一些设计策略。

这个应用的功能就是把银行或经纪商的外汇买入价 / 卖出价公布给交易商。第一个术语就是外汇，说白了就是货币的买卖。它是一个全球范围的分散市场：呃，又一个术语。分散市场意味着货币的交易场所不是唯一的，不像股票交易，只能在一个地方买卖一家公司的

股票（这不是很确切，一些大公司会将他们的股票放到二到三个证券交易所）。

外汇经纪商是做生意的，不会像交易商那样通过观察货币波动的方式赚钱，他们通过价差（有时是佣金）赚钱。价差是买入价和卖出价之间的差额，买入价是这两个价格中较低的那个，它是经纪商为买货币愿意出的价钱，也是你卖掉手里的货币（如果你有的话）能得到的钱；卖出价稍微高一些，是经纪商愿意出售货币的价钱，也是你想买进货币所需要支付的钱。

外汇市场是一个全球市场。纽约股票市场只会在纽约时区的工作时间营业，但世界各地的人们一天到晚都想买卖货币。它是一个 24/5 [1] 市场。按照惯例，它在纽约当地时间星期日下午 5 点（这是新西兰一个工作周的开始）开始营业，纽约当地时间星期五下午 5 点歇业。

主要的交易货币有（括号中是对应的首字母缩写）：美元（USD）、欧元（EUR）、日元（JPY）、英镑（GBP）、澳元（AUD）、加元（CAD）以及瑞士法郎（CHF）。一般情况下，一个经纪商会列出 6 到 40 个外汇交易组合（也叫*外汇对*）。

所有这些对我们来说意味着什么呢？

- 需要从服务端向客户端发送两个价格，以及一个时间戳。
- 需要处理多个外汇对。
- 需要以最小的延迟来处理（突然的波动和过时的价格信息会使交易商蒙受经济损失）。
- 我们的应用需要连续运行 120 小时，然后闲置 48 小时，如此循环往复。

## 3.2 后端

本章的后端示例远比第 2 章的复杂。我们需要多重数据服务（也就是外汇对），如果想给老板留下深刻的印象，不妨称之为*多路技术*。这个应用要能用来做可重复的测试，有看上去很逼真的数据，还要对所有连接的客户端都是同步的，并且不使用任何数据库。要求真高！但这能办到，以下是会用到的技术。

- 单行的 JSON 协议。
- 随机种子。一个指定的随机种子总是能生成一个相同的数据流。这里用它为每一个外汇对生成一个完全可预测的数据集。
- 允许客户端指定随机种子，使客户端可以重复地请求相同的测试数据。
- 将不同时长的周期循环加在一起，再结合一点随机噪声来产出数据，这使得数据看起来更贴近现实（本书不是讨论随机行走和有效市场理论的地方，如果你对那个主题感兴趣，可以找个经济学家聊聊）。
- 检测时间偏差并矫正。

注 1：每周 5 天、每天 24 小时的营业时间。——译者注

## 易测性设计

任何系统在考虑到测试的情况下都有两种设计方式。第一种是不考虑易测性，第二种是使它容易测试，这通常需要费些周折，因为经常需要添加额外的变量和函数。

但是，在易测原则下设计的系统不仅是要容易测试，还要**更快测试**，在极端情况下，这种测试速度差异就像调用一个取值器（几毫秒就能完成测试）和运行一个引入了屏幕抓取和 OCR（光学字符识别）的极度复杂解决方案（需要耗费上千毫秒）的区别。这还会产生连锁反应：测试运行越快，测试的次数就越多，就能在更短的时间内更快地发现 bug，产品就能以更好的质量更快地发布。如果测试包可以每 5 分钟运行一次，当它中断时，你能很快查出代码的问题所在；反之，如果测试包运行时间很长以至于只能在周末运行，你周一早上过来，可能要到周二才能找出是上周哪次修改引入的问题。（这种复杂的测试方案往往很**脆弱**，比如对布局上的细微改动都很敏感）。

在我们的案例中，系统吐出随机数据（好吧，是伪随机数据），此处的**易测性设计**意为控制随机顺序，以便当需要时可以精确地重复，这是被称为**参数注入**的测试设计模式。

在复杂情况下，可能不只是要把 CPU 和内存考虑在内，还要考虑网络，所以每次测试的运行时间可能会相差很多，而返回的 JSON 中包含了精确到毫秒的时间戳，因此，需要想办法确保这些时间戳也是可复验的，正文中有关于如何处理的阐述（如果不做这些，只是对接收到的数据进行字段范围的检查，比如确保每个时间戳的格式正确并且比前一个时间戳晚，确保价格都在 95.00 到 105.00 之间等。这毫无意义，并且会导致遗漏一些不易发现的 bug，从而导致版本回滚）。

我们要做的第一个设计策略，是将消息以 JSON 字符串的形式传输，并严格按照一行一个 JSON 字符串、每条消息一行的格式返回数据。这个设计策略很合理，因为 JSON 是一种灵活并且层次分明的数据格式，在后面的章节中会看到，这种一行一条信息的策略能够使我们的代码更加容易地适用于不支持 SSE 的浏览器。



如果你读了 3.1 节中关于金融行业的部分，就会知道应用同时要公布买入价和卖出价，我故意选择了这样做，而不是仅仅发送一个价格，因为这会更难一点。如果服务端只需发送一个价格，设计策略会更简单，但如果需要再加一个价格，就需要做大量的重构。按支持两个价格的数据来设计，只要简单修改一下代码就能支持 N 个价格的数据，对只有一个价格的数据也能支持得很好。

图 3-1 是后端的主循环流程图（如在第 2 章中看到的，是一个有意的无限循环）。

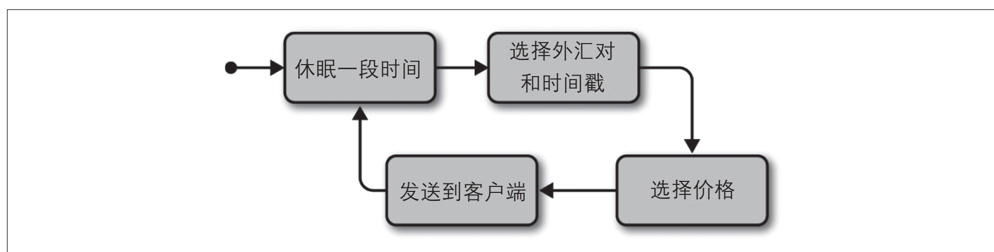



图 3-1：后端的主循环

进入循环之前，有几步初始化工作要做：定义一个类，创建测试用的外汇对，处理客户端输入的参数，设置 Content-Type 数据头。下面是脚本的第一个草稿，使用了硬编码的价格（这样在这个阶段唯一要做的初始化就是设置请求头），代码如下所示：

```
<?php
header("Content-Type: text/event-stream");

while (true) {
    $sleepSecs = mt_rand(250, 500) / 1000.0;
    usleep($sleepSecs * 1000000);

    $d = array(
        "timestamp" => gmdate("Y-m-d H:i:s"),
        "symbol" => "EUR/USD",
        "bid" => 1.303,
        "ask" => 1.304,
        );
    echo "data:" . json_encode($d) . "\n\n";
    @ob_flush(); @flush();
}
```

建议先用下面的命令在命令行运行这段脚本，而不是尝试通过一个 SSE 连接来调试。

```
php fx_server.hardcoded.php
```

这就是 SSE 协议的美妙之处：它是一个简单的文本协议。按下 Ctrl-C 停止运行脚本，会看到这样的输出结果：

```
data:{"timestamp":"2014-02-28 06:09:03","symbol":"EUR\/USD","bid":1.303,
    "ask":1.304}

data:{"timestamp":"2014-02-28 06:09:04","symbol":"EUR\/USD","bid":1.303,
    "ask":1.304}

data:{"timestamp":"2014-02-28 06:09:08","symbol":"EUR\/USD","bid":1.303,
    "ask":1.304}
```

注意，EUR/USD 里的 / 在 JSON 中被转义了。另外，由于 gmdate 被调用，我们看到的输出结果里都是 GMT 时间戳。这是个好习惯，总是以 GMT 格式输出并保存时间数据，如果想以用户的当地时区格式显示，在客户端上调整即可。

## JSON/SSE 协议的开销

用 JSON 格式传输数据会多少开销呢？用 JSON 格式和用最简单的 CSV 格式（data:2014-02-28 03:15:24,EUR/USD,1.303,1.304）相比会相差多少呢？SSE 协议本身又有多少开销呢？

最后一个问题很简单，SSE 的开销是每条消息 6 字节，也就是“data:”和额外的换行符。这可以看做是第 6 章和第 7 章中介绍的向后兼容方案。

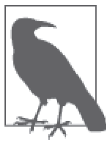
JSON 字符串可以不用这么长，出于易读的目的我采用了冗长的字段名，不然它也可以像这样：

```
data:{"t":"2014-02-28 06:09:03","s":"EUR\/USD","b":1.303,"a":1.304}
```

如果是二进制的协议会怎样呢？不论 JavaScript 还是 SSE 都不适合用二进制，暂且不管这个，时间戳需要 4 字节（即便需要精确到毫秒，或者需要用到 2030 年，最多也就需要 8 字节），外汇对需要 7 字节加一个零终止符，买入价 / 卖出价分别需要 8 字节，一共是 28 字节（假设一条数据的结束符是隐性的）。表 3-1 对此进行了概括。

表3-1：不同数据格式所需的字节数对比

| | 使用SSE | 使用向后兼容 |
|---|---|---|
| 二进制 | 34 | 28 |
| CSV | 46 | 40 |
| 简短的 JSON | 69 | 63 |
| 易读的 JSON | 86 | 80 |



因为我们会立刻清除数据（以最大限度降低延迟），你可能想把每条信息的 TCP/IP 包和以太网络帧的开销也包含进来。如果和一个轮询方案比较，这可能是合理的，比如，以平均一秒一条消息的频率推送数据，那就会比一分钟一次的轮询多出 59 倍的 TCP/IP 数据包，如果涉及 WiFi 和移动网络的情况，这个差距可能还会更大。但是如果用轮询（尤其是长轮询，见第 6 章），不要忘了每个方向（服务端到客户端和客户端到服务端）的请求都要把 HTTP 请求头考虑在内。记住，cookie 和鉴定文件头是随着每次请求发送的。

正如第 1 章中提到的，要对两种替代方案做一个有效的对比，我认为最好的方式是在尽可能真实的负载条件下搭建这两种方案，然后以同样的标准进行测试对比。**真实的**也意味着服务器和测试客户端应该处在不同的数据中心，除非是在做一个公司内网的应用。

在以表 3-1 的数据为基础做出决策之前，请记住，SSE 通信数据能够并且应该用 GZIP 格式压缩，数据越紧凑，GZIP 能压缩的量就越少。

我们的外汇数据做得很漂亮并且有规律，所以你可能会打算用 CVS 格式取代 JSON 格

式。我会继续用 JSON，因为在其他应用中数据可能不会这么简单（JSON 能处理嵌套数据结构），并且新增一个字段也很简单。事实上，随着该应用的展开，会引入更复杂的数据结构。我将坚持使用易读的字段名，以帮我们保持思路清晰。

在最初的 fx_server.hardcoded.php 中，实现了后端流程中 3 个高级步骤中的 2 步：休眠和发送数据到客户端。在下一节中，会实现对外汇对和价格的选择，而不用对它们硬编码。

## 3.3　前端

后面会进一步完善后端代码，但既然现在已经有了一个最简单的服务端脚本。接下来就是要创建一个最简单的 HTML 页面，代码如下：

```
<!doctype html>
<html>
    <head>
    <meta charset="UTF-8">
    <title>FX Client: latest prices</title>
    </head>
    <body>

        <table border="1" cellpadding="4" cellspacing="0">
            <tr>
                <th>USD/JPY</th>
                <th>EUR/USD</th>
                <th>AUD/GBP</th>
            </tr>
            <tr>
                <td id="USD/JPY"></td>
                <td id="EUR/USD"></td>
                <td id="AUD/GBP"></td>
            </tr>
            </table>

    <script>
    var es = new EventSource("fx_server.hardcoded.php");
    es.addEventListener("message", function (e) {
        var d = JSON.parse(e.data);
        document.getElementById(d.symbol).innerHTML = d.bid;
        }, false);
    </script>

    </body>
</html>
```

在浏览器中访问这个页面，会看到一个有 3 列的表格，中间 EUR/USD 这一列的单元格里会出现 1.303，仅此而已。这非常枯燥无味，对吧？但是，服务端确实在重复地发送 1.303。这段前端代码，虽然很基础，但将在后端做的每一个优化都有效。

如果你看过第 2 章，上面前两行 JavaScript 代码应该看起来很熟悉，创建一个 EventSource

对象，指定要连接的服务器地址，然后把包含了一段 JSON 字符串的 e.data 赋值给 message 事件处理程序，所以事件处理程序的第一行是 var d = JSON.parse(e.data);[2]，用以把字符串转化成 JavaScript 对象。



如果 JSON 数据有问题，JSON.parse 会抛出一个异常。从第 5 章开始，我们会把它包在 try{} 和 catch(e){} 块中，作为使这段代码达到产品级品质所采取的措施之一。

事件处理程序中的第二行前半部分从 document.getElementById(d.symbol) 开始，用以找到 HTML 中对应的表格单元格，它是这些 id="USD/JPY"、id="EUR/USD"、id="AUD/GBP"[3] 中的一个。后半部分代码 .innerHTML = d.bid 用以将买入价填充到前面获取的单元格中。

我们在后面还会讨论前端的问题，但现在先回过头来看一下后端代码。

## 3.4 可复现的真实随机数据

之前我们创建了一个生成可复现数据的脚本，现在要让这些数据具备随机性和真实性。fx_server.hardcoded.php 的第一个问题是仅有一个外汇对，但我们需要不同的外汇对（即货币组合）。鉴于外汇对之间有很多共性，只是数值不同，我们创建了 FXPair 类，如下面的代码所示。如果对 PHP 的类不熟，可以参考附录 C.1 节。

```
<?php
class FXPair {
    /** 外汇对名称 */
    private $symbol;
    /** 买入价基准值 */
    private $bid;
    /** 价差。与 $bid 相加得到 " 卖出价 " */
    private $spread;
    /** 价格精确到的小数点位数 */
    private $decimalPlaces;
    /** 一次大循环的时长秒数 */
    private $longCycle;
    /** 一次小循环的时长秒数 */
    private $shortCycle;

    /** 构造函数 */
```

注 2：支持 SSE 的浏览器都支持 JSON.parse，但是，在讨论针对旧版浏览器的向后兼容方案时，会发现有些旧版浏览器实在太落后了，无法支持 JSON.parse，特别是 IE6/IE7。不过要修复这个问题并不难。

注 3：HTML5 中，DOM 的 ID 可以包含除空格外的任何字符，但是，如果你的代码要兼容 IE7 或者 IE8 等 HTML4 浏览器，你就需要修改一下数据中这些外汇对的名称，比如，要把所有的“/”替换成“_”，DOM 的 ID 就会变成 "USD_JPY"、"EUR_USD" 等。（还要确保 ID 不是以数字开头，在 IE6 中，还要确保不是以下划线开头）。

```
    public function __construct($symbol, $b, $s, $d, $c1, $c2) {
        $this->symbol = $symbol;
        $this->bid = $b;
        $this->spread = $s;
        $this->decimalPlaces = $d;
        $this->longCycle = $c1;
        $this->shortCycle = $c2;
    }

    /** @param int $t 从 1970 年到现在的秒数 */
    public function generate($t) {
    $bid = $this->bid;
    $bid += $this->spread * 100 *
        sin((360 / $this->longCycle) * (deg2rad($t % $this->longCycle)));
    $bid += $this->spread * 30 *
        sin((360 / $this->shortCycle) * (deg2rad($t % $this->shortCycle)));
    $bid += (mt_rand(-1000, 1000) / 1000.0) * 10 * $this->spread;
    $ask = $bid + $this->spread;

    return array(
        "timestamp" => gmdate("Y-m-d H:i:s", $t),
        "symbol" => $this->symbol,
        "bid" => number_format($bid, $this->decimalPlaces),
        "ask" => number_format($ask, $this->decimalPlaces),
        );
    }
}
```

该类有买入价、价差、精确位数这些成员变量。bid 保存买入价基准值：买入价的值会在这个基础上波动。spread 是买入价和卖出价之间的价差（参见 3.1 节）。为什么有一个精确位数成员变量？因为一般涉及日元（JPY）的货币价格是精确到小数点后 3 位，其他货币的是小数点后 5 位。

还有两个成员变量：long_cycle 和 short_cycle。在 generate 函数中会看到它们控制着价格波动的速度。使用两个周期的目的是让周期性表现看上去更有趣一点。长周期的权重是 100，短周期的权重是 30，除此之外，还加入了一些随机噪声，权重是 10。你是不是想问 (mt_rand(-1000,1000)/1000.0)？ PHP 没有生成随机浮点数的函数，所以这里创建了一个从 –1000 到 +1000（含 –1000 和 +1000）的随机整数，然后除以 1000 来获得的一个从 –1.000 到 +1.000 的随机浮点数，每次都乘以价差和权重。



关于为什么用 mt_rand 以及随机种子怎么设置的问题，参见附录 C.2 节。

最后，generate 返回一个关联数组（即 JavaScript 中的对象、.NET 中的字典、C++ 中的映射）。number_format 用来裁掉多余的小数点位数，所以，98.1234545984 变成了 98.123。

怎么使用这个类呢？在 fx_server.seconds.php 的最上面为每一个外汇对创建了一个对象（EUR/USD 出现了两次，因为我们想让它以两倍于其他外汇对的频率更新）：

```
$symbols = array(
    new FXPair("EUR/USD", 1.3030, 0.0001, 5, 360, 47),
    new FXPair("EUR/USD", 1.3030, 0.0001, 5, 360, 47),
    new FXPair("USD/JPY", 95.10, 0.01, 3, 341, 55),
    new FXPair("AUD/GBP", 1.455, 0.0002, 5, 319, 39),
    );
```

接下来，在主循环中随机获取一个要修改的外汇对：

```
$ix = mt_rand(0,count($symbols)-1);
```

然后，在 fx_server.hardcoded.php 中把硬编码的 `$d` 数组替换成调用 `generate`：

```
$d = $symbols[$ix]->generate($t);
```

完整的 fx_server.seconds.php 代码如下所示：

```
<?php
include_once("fxpair.seconds.php");

header("Content-Type: text/event-stream");

$symbols = array(
    new FXPair("EUR/USD", 1.3030, 0.0001, 5, 360, 47),
    new FXPair("EUR/USD", 1.3030, 0.0001, 5, 360, 47),
    new FXPair("USD/JPY", 95.10, 0.01, 3, 341, 55),
    new FXPair("AUD/GBP", 1.455, 0.0002, 5, 319, 39),
    );

while (true) {
    $sleepSecs = mt_rand(250, 500) / 1000.0;
    usleep($sleepSecs * 1000000);

    $t = time();
    $ix = mt_rand(0, count($symbols) - 1);
    $d = $symbols[$ix]->generate($t);
    echo "data:" . json_encode($d) . "\n\n";
    @ob_flush();@flush();
    }
```

注意一下关于这段代码的几个细节，生成的价格只是以当前时间为基础，而不会保存先前的值，然后在此基础上随机地加 / 减，这可能也是你首先想到的实现随机价格的方案。这段代码不仅漂亮整洁，还可以进行可复现的可靠测试，而且还会带来一个好处：可以在 `$symbols` 数组中放两个 EUR/USD 的 FXPair 对象，从而获取两倍于其他外汇对的价格数据。

为什么用 `usleep()` 而不是 `sleep()`？参见附录 C.6 节。

你是不是在想，既然 `$t` 只是用来传给 `generate()` 的，为什么还要在主循环中给它赋值，

而没有把 `$t = time();` 放在 `generate()` 里？这要回到 3.2 节的附注内容“易测性设计”：通过使用一个参数，我们可以传入某个确定的值，`generate()` 则总是能返回一个相同的结果，这样能很容易创建 `generate()` 的单元测试；如果不这么做，全局函数 `time()` 就成了 `generate()` 函数的一个附属品。而这会让人感觉很不舒服。[“很不舒服”是从 100 多页的 *xUnit Test Patterns*（Gerard Meszaros 著，Addison-Wesley 出版社出版）一书中总结出来的结论。如果你想更深入地了解这一点，可以看看那本书。]

## 3.5 精磨时间戳

在命令行运行 fx_server.seconds.php，会看到这些输出：

```
data:{"timestamp":"2014-02-28 06:49:55","symbol":"AUD\/GBP","bid":"1.47219",
    "ask":"1.47239"}

data:{"timestamp":"2014-02-28 06:49:56","symbol":"USD\/JPY","bid":"94.956",
    "ask":"94.966"}

data:{"timestamp":"2014-02-28 06:49:56","symbol":"EUR\/USD","bid":"1.30931",
    "ask":"1.30941"}

data:{"timestamp":"2014-02-28 06:49:57","symbol":"EUR\/USD","bid":"1.30983",
    "ask":"1.30993"}

data:{"timestamp":"2014-02-28 06:49:57","symbol":"EUR\/USD","bid":"1.30975",
    "ask":"1.30985"}

data:{"timestamp":"2014-02-28 06:49:57","symbol":"AUD\/GBP","bid":"1.47235",
    "ask":"1.47255"}

data:{"timestamp":"2014-02-28 06:49:58","symbol":"AUD\/GBP","bid":"1.47129",
    "ask":"1.47149"}
```

数据看起来很漂亮并且是随机的，对吧？但如果看久了，就会发现这是我们之前输入的长周期和短周期程序指令。注意看 EUR/USD 有两个时间戳一样的数据，接下来要做的就是把毫秒加到时间戳中。

只需要这样改一下代码：

(1) 在主循环中，用 `microtime(true)` 替代 `time()`；
(2) 在 `generate()` 中，把毫秒加到已经格式化的时间戳里。

`microtime(true)` 返回一个浮点数：从 1970 年到当前时间戳的秒数（和 `time()` 一样），但精确到了毫秒。

怎么来格式化时间戳？现在是这样的：

```
'timestamp'=>gmdate("Y-m-d H:i:s",$t),
```

这个仍然管用，即使 `$t` 是一个浮点数，它仍然是从 1970 年到现在的秒数，PHP 会为 `gmdate()` 函数隐式地把它转化成整数。所以只需要把毫秒数加上去。

可以通过 `($t*1000)%1000` 获取毫秒数（先乘以 1000 将 `$t` 变成从 1970 年到现在的毫秒数，然后取最后 3 位数），然后用 `sprintf` 对它进行格式化，这样它就会总是 3 位数字的格式，并且前面还会有一个小数点：

```
'timestamp'=>gmdate("Y-m-d H:i:s",$t).sprintf(".%03d",($t*1000)%1000),
```

以下是新版 FXPair 类的全部代码：

```
<?php
class FXPair {
    /** 外汇对的名称 */
    private $symbol;
    /** 买入价基准值 */
    private $bid;
    /** 价差，与 $bid 相加得到 " 卖出价 " */
    private $spread;
    /** 价格的精确小数点位数 */
    private $decimalPlaces;
    /** 一次大循环的时长秒数 */
    private $longCycle;
    /** 一次小循环的时长秒数 */
    private $shortCycle;

    /** 构造函数 */
    public function __construct($symbol, $b, $s, $d, $c1, $c2) {
        $this->symbol = $symbol;
        $this->bid = $b;
        $this->spread = $s;
        $this->decimalPlaces = $d;
        $this->longCycle = $c1;
        $this->shortCycle = $c2;
    }

    /** @param float $t 从 1970 年到现在的秒数，精确到毫秒 */
    public function generate($t) {
    $bid = $this->bid;
    $bid += $this->spread * 100 *
        sin((360 / $this->longCycle) * (deg2rad($t % $this->longCycle)));
    $bid += $this->spread * 30 *
        sin((360 / $this->shortCycle) * (deg2rad($t % $this->shortCycle)));
    $bid += (mt_rand(-1000, 1000) / 1000.0) * 10 * $this->spread;
    $ask = $bid + $this->spread;
    return array(
        "timestamp" => gmdate("Y-m-d H:i:s", $t) .
            sprintf(".%03d", ($t * 1000) % 1000),
        "symbol" => $this->symbol,
        "bid" => number_format($bid, $this->decimalPlaces),
        "ask" => number_format($ask, $this->decimalPlaces),
        );
    }
}
```

下面这段 fx_server.milliseconds.php 脚本用到了它：

```
<?php
include_once("fxpair.milliseconds.php");

header("Content-Type: text/event-stream");

$symbols = array(
    new FXPair("EUR/USD", 1.3030, 0.0001, 5, 360, 47),
    new FXPair("EUR/USD", 1.3030, 0.0001, 5, 360, 47),
    new FXPair("USD/JPY", 95.10, 0.01, 3, 341, 55),
    new FXPair("AUD/GBP", 1.455, 0.0002, 5, 319, 39),
    );

while (true) {
    $sleepSecs = mt_rand(250, 500) / 1000.0;
    usleep($sleepSecs * 1000000);

    $t = microtime(true);
    $ix = mt_rand(0, count($symbols) - 1);
    $d = $symbols[$ix]->generate($t);
    echo "data:".json_encode($d) . "\n\n";
    @ob_flush();@flush();
    }
```

运行 fx_server.milliseconds.php 时，会看到这样的输出结果：

```
data:{"timestamp":"2014-02-28 06:49:55.081","symbol":"AUD\/GBP",
    "bid":"1.47219","ask":"1.47239"}

data:{"timestamp":"2014-02-28 06:49:56.222","symbol":"USD\/JPY",
    "bid":"94.956","ask":"94.966"}

data:{"timestamp":"2014-02-28 06:49:56.790","symbol":"EUR\/USD",
    "bid":"1.30931","ask":"1.30941"}

data:{"timestamp":"2014-02-28 06:49:57.002","symbol":"EUR\/USD",
    "bid":"1.30983","ask":"1.30993"}

data:{"timestamp":"2014-02-28 06:49:57.450","symbol":"EUR\/USD",
    "bid":"1.30972","ask":"1.30982"}

data:{"timestamp":"2014-02-28 06:49:57.987","symbol":"AUD\/GBP",
    "bid":"1.47235","ask":"1.47255"}

data:{"timestamp":"2014-02-28 06:49:58.345","symbol":"AUD\/GBP", "bid":"1.47129","a
    sk":"1.47149"}
```

在本书的源码里，有一个叫 fx_client.basic.milliseconds.html 的文件，它使你能够在浏览器中看到如图 3-2 所示的结果。每次运行这个脚本时，都会看到 3 个货币价格的起落，如果你有盯着油漆变干[4]的嗜好，应该会看得很享受。这也有利于人工测试，只要你不介意至

注 4：“盯着油漆干”，原文“watching paint dry”，形容一件很漫长而无趣的事，就像盯着新刷的油漆等它变干一样。——译者注

少盯着它看 6 分钟（长周期的时长）。但是每次运行这个脚本，外汇对的价格，出现的顺序，当然还有时间戳，都会不一样。回顾一下 3.2 节的附注内容“易测性设计”，想想为什么我们想要对它做出改进。

| USD/JPY | EUR/USD | AUD/GBP |
|---|---|---|
| 94.628 | 1.30016 | |

图 3-2：运行了几秒之后的 fx_client.basic.milliseconds.html

## 3.6　控制好随机性



本章接下来的内容都是关于后端优化的。如果你对前端更感兴趣，可以略过并直接去看第 4 章。

试一下在 fx_server.milliseconds.php 的最上面加上这一行：`mt_srand(123);`。它将随机种子设成了指定的值。

停止运行这段脚本，然后再重新运行，注意到了什么？如果你原本以为设置随机种子会产生一个可复现的结果，那现在你一定震惊了：一切都发生了变化。但仔细看，会看到那些跳动着的外汇对的顺序是不变的，EUR/USD 跳 3 次，然后 USD/JPY 跳 3 次，然后 AUD/GBP 跳 3 次，然后 USD/JPY 跳 3 次[5]。这很有意义，因为控制下一个外汇对符号的代码 `$ix = mt_rand(0,count($symbols)-1)` 是一个简单的随机代码。

如果你看得非常仔细，会发现每两个时间戳之间的差值几乎是一样的。比如，在一次运行中相差 431 毫秒，再次运行相差 430 毫秒，第三次运行还是相差 431 毫秒。这也是有意义的，因为两次跳动之间的时间间隔也是一个简单的随机代码：`$sleepSecs=mt_rand(250,500)*1000;`。计时的差异是因为 CPU 的速度，即服务器当时的忙碌程度以及地球另一边有只蝴蝶在振动翅膀造成的。

但是为什么价格会不同呢？因为它们都是以 `$t`（服务器当前的时间）为基础，再加入了一点随机的噪声。因为，我们需要对 `$t` 加以控制。现在，你的第一想法是不是这样：在运行每个单元测试之前，修改一下系统时间？我喜欢你的范儿。当我们需要穿过一面墙而手里只有一把大锤的时候，希望有你在身边。老实说，我也这么想过。

注 5：指定随机种子后，确切的随机顺序会因 PHP 的版本不同而不同，可能还会与操作系统版本有关，写作本书时，我用的是 64 位 Linux 的 PHP 5.3。

但在这种情况中，穿过那堵墙有更简单的方式：从门穿过去。而且那还是我们之前就已经放在那儿的。我说的是把 `$t` 传给 `generate()`，而不是让 `generate()` 自己调用 `microtime(true)`。

来感觉一下，用 `$t=1234567890.0;` 替换 `$t = microtime(true);`。现在输出结果如下所示：

```
data:{"timestamp":"2009-02-13 23:31:30.000","symbol":"EUR\/USD",
    "bid":"1.31103","ask":"1.31113"}
```

这样修改之后，每次运行脚本，$t 的值都是一样的，与 CPU、负载或昆虫行为无关。

我们当然不想让它一直都是 2009 年 2 月 13 日。下面是我们代码的下一个版本，提供了控制 `$t` 的选项：

```
<?php
include_once("fxpair.milliseconds.php");

header("Content-Type: text/event-stream");

$symbols = array(
    new FXPair("EUR/USD", 1.3030, 0.0001, 5, 360, 47),
    new FXPair("EUR/USD", 1.3030, 0.0001, 5, 360, 47),
    new FXPair("USD/JPY", 95.10, 0.01, 3, 341, 55),
    new FXPair("AUD/GBP", 1.455, 0.0002, 5, 319, 39),
    );

if (isset($argc) && $argc >= 2)
    $t = $argv[1];
elseif (array_key_exists("seed", $_REQUEST))
    $t = $_REQUEST["seed"];
else {
    $t = microtime(true);
    echo "data:{\"seed\":$t}\n\n";
}
mt_srand($t * 1000);

while (true) {
    $sleepSecs = mt_rand(250, 500) / 1000.0;
    usleep($sleepSecs * 1000000);
    $t += $sleepSecs;

    $ix = mt_rand(0, count($symbols) - 1);
    $d = $symbols[$ix]->generate($t);
    echo "data:" . json_encode($d) . "\n\n";
    @ob_flush(); @flush();
    }
```

和 fx_server.milliseconds.php 相比，最大的变化是主循环前面的那段代码块。但事实上，这段代码很普通，如果从命令行（`if(isset($argc)...`）中运行脚本，它会获取命令行的第一个参数作为 `seed` 的值；如果在浏览器中运行，它会查找名为 `seed` 的参数[6]并使用那个（`$_REQUEST['seed'];`）；如果前两者都没有设置，就使用当前时间来初始化，然后它会输出一行信息，表明它使用的是哪个 `seed`。这样的话，如果出现什么问题，

可以用这个随机种子重新生成数据流。获得随机种子之后，就调用 mt_srand。把 $t 乘以 1000，mt_srand 会把它截断成整数，这就是我们所说的精确到毫秒，而不是精确到微秒。

主循环中的修改很简单。将 $t=microtime(true); 从循环的开头移除，然后在循环末尾给 $t 加上了休眠的秒数。换句话说，如果 $t 是 1234567890.0，也就是模拟当前时间是 2009-02-13 23:31:30.000，然后休眠 0.325 秒，更新之后模拟的当前时间就是 2009-02-13 23:31:30.325。

## 3.7 为时间的真正流逝留出余地

多么有趣的小标题！从单元测试的角度来说，上一节末尾的那段代码已经足够好了。但是，你是否试过不用随机种子来运行？为了一探究竟，我把下面这段代码[7]加到了 echo "data:"... 上面：

```
$now=microtime(true);
echo ":".
    gmdate("Y-m-d H:i:s",$now).
    sprintf(".%03d",($now*1000)%1000).
    "\n";
```

在一行前面加一个冒号是 SSE 里的注释方式。在浏览器中看不到注释，所以在命令行中运行这个脚本。刚开始时 $now 和 $t 是同步的，但是跳动几次之后，$now 可能会慢几毫秒。去烧一壶水吧，回来的时候这个差值会变成几百毫秒。运行 24 小时的话会差几分钟。（顺便说一下，这个问题在你给定一个种子时也会存在，只是更难看出。）

好吧，那只是测试数据，确实不是很重要，但调整休眠时间以使它匹配真实的时间流逝，可能是你必备的一项技能。所以，我们快点行动吧。

这里会用一个变量 $clock 来保存服务器时间。在脚本开始时会把服务器当前时间赋给 $clock，但真正发挥作用的修改是在主循环中。$now=microtime(true); 又回来了！然后通过 $adjustment = $now - $clock; 来计算时间的偏差。这里的关键概念是，要休眠时，让休眠时间比原来想要的少一点：

```
usleep( ($sleepSecs - $adjustment) * 1000000);
```

$t 还是像前面一样更新，即加上 $sleepSecs，不需要使用 $adjustment，但是当以相同的方式更新 $clock，$clock 代表着理想情况下（假设处理器运行得无穷快）的服务器时间。

---

注 6：是的，这里故意用的 $_REQUEST，这样就可以从 GET、POST，甚至 cookie 中获取数据，在这里，能够通过 cookie 设置随机种子是个特性，而不是 bug！ 更多关于 PHP 超全局变量的信息参见附录 C “超全局变量”。

注 7：可以在本书源码的 fx_server.repeatable_with_datestamp.php 文件中找到这段代码。

fx_server.adjusting.php 文件的完整代码如下所示，从本书的源代码中也可以找到 fx_server.adjusting_with_datestamp.php，它还是用 SSE 注释来显示伪造数据刚好以和真实时间流逝一样的节奏吐出。你还会发现 fx_client.basic.adjusting.html 和 fx_client.basic.adjusting123.html，前者与 fx_server.adjusting_with_datestamp.php 相连（该版本显示了所选的随机种子），后者设置了一个显式的随机种子，因此在每次重载时都能显示可复现的数据。

```
<?php
include_once("fxpair.milliseconds.php");

header("Content-Type: text/event-stream");

$symbols = array(
    new FXPair( ‘EUR/USD’ , 1.3030, 0.0001, 5, 360, 47),
    new FXPair( ‘EUR/USD’ , 1.3030, 0.0001, 5, 360, 47),
    new FXPair( ‘USD/JPY’ , 95.10, 0.01, 3, 341, 55),
    new FXPair( ‘AUD/GBP’ , 1.455, 0.0002, 5, 319, 39),
    );
$clock = microtime(true);
if (isset($argc) && $argc >= 2)
    $t = $argv[1];
elseif (array_key_exists( ‘seed’ , $_REQUEST))
    $t = $_REQUEST[ ‘seed’ ];
else {
    $t = $clock;
    echo "data:{\"seed\":$t}\n\n";
    }
mt_srand($t * 1000);

while (true) {
    $sleepSecs = mt_rand(250, 500) / 1000.0;
    $now = microtime(true);
    $adjustment = $now - $clock;

    usleep(($sleepSecs - $adjustment) * 1000000);
    $t += $sleepSecs;
    $clock += $sleepSecs;

    $ix = mt_rand(0, count($symbols) - 1);
    $d = $symbols[$ix]->generate($t);
    echo "data:" . json_encode($d) . "\n\n";
    @ob_flush(); @flush();
    }
```

# 3.8 本章内容盘点

本章讨论的范围很广。结合易测性设计原则，我们一步一步地设计了一个提供随机数据的后端（同时也了解了外汇市场是如何运作的），然后用 SSE 把数据推送到客户端。但我们推进得太快了，所以下一章的开头会做一些重构，然后增加一些关于数据存储特性的内容。

# 第 4 章
# 别安于现状

现在已经做得很好了，我们有一个相当复杂但又相对易测的服务端，还有一个可以看到其工作的基础前端。也差不多到了该恢复平衡、优化前端的时候了，但在把注意力转移到前端之前，还需要在后端上做些改变：修改数据结构，并因此不再兼容之前见过的那些 fx_client.basic.*.html 文件。

## 4.1 数据的更多构成

现在一条数据就一项 JSON 记录，要做的主要修改就是允许传输多行数据，同时还有两个“请求头”字段：一个是外汇对的名称，一个是服务器时间戳。所以，数据结构会变成这样：

```
symbol:string
timestamp:string ("YYYY-MM-DD HH:MM:SS.sss")
rows:array
```

并且 `rows` 里的每一项数据都是这样的结构：

```
timestamp:string ("YYYY-MM-DD HH:MM:SS.sss")
bid:double
ask:double
```

为什么要做这些？一个理由是为可能需要发送数组数据（比如，支持历史数据请求）做好准备。当然，可以把每一行数据作为一行单独的 JSON 数据发送，这样会使数据量增加几字节，可能是每行增加 12 字节。更好的理由是可以告诉客户端这是一个逻辑块，现在服务端每发送一条 SSE 消息，事件回调函数就会被调用一次，应用可能就要更新一次视图。如果把几百行的数据放到一个块里发送，客户端就可以把它们作为一个数据块进行处理，

然后只需在处理完之后更新一次视图。

另一个理由是这样更灵活，可以新增一种类型字段，改变 `rows` 的含义，比如增加一个字段说明它是 GZIP 压缩过的 CSV 格式的数据，而不是 JSON 数组，还可以增加一个版本号字段，谁知道我们将来还想干嘛呢？[1]

说了这么多，代码要做的改动其实很小，只会影响 FXPair 类中的 `generate()` 方法。相对于 fxpair.milliseconds.php，fxpair.structured.php 中 `generate()` 方法的第二部分像下面这样：

```
$ts = gmdate("Y-m-d H:i:s", $t) . sprintf(".%03d", ($t*1000)%1000);
return array(
    "symbol" => $this->symbol,
    "timestamp" => $ts,
    "rows" => array(
        array(
            "timestamp" => $ts,
            "bid" => number_format($bid, $this->decimal_places),
            "ask" => number_format($ask, $this->decimal_places),
        )
    )
);
```



在 PHP 中，一个含有带名称的键的数组称为关联数组，相当于 JSON 格式的对象。没有键（就像上面的代码那样）或有数值型键的数组，相当于 JSON 格式的数组。

注意，数据的时间戳和消息的时间戳设成一样的了，但它们并不必须是一样的，数据行里的时间戳应该来自证券交易所并且有官方的交易时间戳，所以它可能比消息的时间戳早几毫秒；如果是历史数据，那可能早几个月甚至几年。

## 4.2 重构PHP

PHP 代码不超过 40 行，所以真的没有多少要重构的。但我敢打赌，你一遍又一遍地看到下面这段代码一定会觉得牙根痒痒：

```
$d = $symbols[$ix]->generate($t);
echo "data:".json_encode($d)."\n\n";
@ob_flush();@flush();
```

所以我把它改成这样：

```
sendData($symbols[$ix]->generate($t));
```

注 1：我们之前已经这样做了，以一种临时的方式，即把所选的 seed 放到发送的 SSE 消息中。

sendData() 的实现很简单：

```
function sendData($data){
echo "data:";
echo json_encode($data)."\n";
echo "\n";
@flush();@ob_flush();
}
```

（拆分成 3 个 echo 语句真的不是为了遵循本书的格式规范，这是为第 6 章将要进行的修改做准备。提示一下：中间那一行才是真正的数据，而“data:”前缀和额外的换行符是 SSE 协议规范所要求的。）

可以在本书源码文件 fx_server.structured.php 中看到这段修改，其他修改只是把包含 fxpair.milliseconds.php 替换成包含 fxpair.structured.php。

## 4.3　重构JavaScript

现在的 JavaScript 一共 6 行，但进一步看，整理一下代码结构是有好处的，这里所做的一些设计策略也是为兼容旧版浏览器做铺垫。

首先定义两个全局变量：

```
var url = "fx_server.structured.php?";
var es = null;
```



为什么在 URL 的末尾放一个问号？后面需要在 URL 后追加一些参数，这样的话在追加参数时不需要知道所追加的是 URL 的第一个参数（前缀必须是 ?）还是之后的（以 & 为前缀）。

现在把创建 EventSource 对象的代码放在 startEventSource() 函数中，如下所示：

```
function startEventSource(){
if(es)es.close();
es = new EventSource(url);
es.addEventListener("message", function(e){processOneLine(e.data);}, false);
es.addEventListener("error", handleError, false);
}
```

下一章会写 handleError 函数，现在就这么写吧：

```
function handleError(e){}
```

接下来把 startEventSource() 的调用封装到 connect 函数中，如下所示：

```
function connect(){
if(window.EventSource)startEventSource();
// 否则在这里处理向后兼容
}
```

你可能听说过这句话：程序设计中的任何问题都可以通过增加一个中间层来解决。好吧，显然我们在增加一个抽象层，那么我们要解决的问题是什么呢？还是为向后兼容：所有连接技术（比如长连接）相关的代码都会放到 `connect()` 里，也包括对应选技术的检测。使用 SSE 的特定代码放在 `startEventSource()` 里。

要让一切运转起来，可在页面加载完成时立即调用 `connect()`，最简单的方式就是把下面这段代码包在一对 `<script>` 标签中，然后放在页面底部：

```
setTimeout(connect, 100);
```

如果你用 jQuery，也许更熟悉下面的方式（而且可以把这段代码放在任何地方，不是必须要放在底部）：

```
$(function(){ setTimeout(connect, 100); });
```



我们用了一个 0.1 秒的延时因为有些版本的浏览器需要这样。比如，Safari 的某些版本，如果不用延时，你可能会看到“加载菊花”一直在转呀转，我讨厌 100 毫秒这个“魔法数字”，但它确实有用。

接下来的重构是把数据处理放到一个专门的函数 `processOneLine(s)` 中，它接收单行 JSON 字符串作为参数：

```
function processOneLine(s) {
var d = JSON.parse(s);
if (d.seed) {
    var x = document.getElementById("seed");
    x.innerHTML += "seed=" + d.seed;
    }
else if (d.symbol) {
    var x = document.getElementById(d.symbol);
    for (var ix in d.rows) {
        var r = d.rows[ix];
        x.innerHTML = r.bid;
        }
    }
}
```

这个函数也展示了如何处理上一节描述的 JSON 格式的变化。这里用一个循环来展示如何处理数据的每一行，在这里，每一行都更新同一个 `div`，所以，实际上只有最后一行数据有用。但大多数时候你会关心所有数据，并且想要一个循环。

看看做了这些重构的 fx_client.basic.structured.html 文件，它和之前的版本（fx_client.basic.adjusting.html）功能完全一样，这是所有良好重构的目标。绝不要将重构和新增功能混在一起做。

顺便说一下，如果真的只需要最后一行数据，上面那一段代码可以改成这样：

```
var x = document.getElementById(d.symbol);
x.innerHTML = d.rows[ d.rows.length - 1 ].bid;
```

能这样做是因为 `d.rows` 是一个数组，不是一个对象。如果 `d.rows` 是对象（比如，以时间戳为键），就只能用循环。说起这个，因为 `d.rows` 是一个数组，所以也可以这样写主循环：

```
for (var ix = 0; ix < d.rows.length; ++ix) {
    var r = d.rows[ix];
    x.innerHTML = r.bid;
    }
```

## 4.4 历史数据存储

在我们的应用中，得到数据当即就用，然后就丢了。如果把接收到的数据都保存下来，那我们就可以做更多事情，比如，可以把最后 5 分钟或者 24 小时的数据做成一个表格，或者做成一个图表。

我们从添加这个全局变量开始，它用于保存所有外汇对历史数据：

```
var fullHistory = {};
```

这是一个 JavaScript 对象，但是会被当成一个关联数组（也叫映射、字典、散列、键值对存储）来用。以外汇对的名称作键，值又是一个关联数组。当接收到一行数据（一条 JSON 字符串，包含了一行或更多行数据），如下所示：

```
if(!fullHistory.hasOwnProperty(d.symbol))
    fullHistory[d.symbol] = {};
```

当某一个外汇对名称（`d.symbol`）第一次出现时，先为它创建一个 JavaScript 空对象，然后，像下面这样来填充这个对象：

```
for (var ix in d.rows) {
     var r = d.rows[ix];
     fullHistory[d.symbol][r.key] = r.value;
     }
```

这段代码片段里假定每一行数据都有 `key` 字段和 `value` 字段。在我们的代码里 `r.timestamp` 是键，值是一个有两个值的数组 `[r.bid,r.ask]`。所以，新的 `processOneLine(s)` 函数的完整代码是：

```
function processOneLine(s) {
var d = JSON.parse(s);
if (d.seed) {
    var x = document.getElementById("seed");
    x.innerHTML += "seed=" + d.seed;
    }
else if (d.symbol) {
    if (!fullHistory.hasOwnProperty(d.symbol))fullHistory[d.symbol] = {};
    var x = document.getElementById(d.symbol);
    for (var ix in d.rows) {
        var r = d.rows[ix];
        x.innerHTML = d.rows[ix].bid;
        fullHistory[d.symbol][r.timestamp] = [r.bid, r.ask];
        }
    update_history_table(d.symbol);
    }
}
```

如果想要用一个 HTML 表格显示历史记录中某一个外汇对最近的 10 条数据，该怎么做呢？下面是为一个外汇对创建表格的函数：

```
function makeHistoryTbody(history) {
var tbody = document.createElement("tbody");
var keys = Object.keys(history).sort().slice(-10).reverse();

var timestamp, v, row, cell;
for (var n = 0; n < keys.length; n++) {
    timestamp = keys[n];
    v = history[timestamp];
    row = document.createElement("tr");
    cell = document.createElement("th");
    cell.appendChild(document.createTextNode(timestamp));
    row.appendChild(cell);
    cell = document.createElement("td");
    cell.appendChild(document.createTextNode(v[0]));
    row.appendChild(cell);
    cell = document.createElement("td");
    cell.appendChild(document.createTextNode(v[1]));
    row.appendChild(cell);
    tbody.appendChild(row);
    }
return tbody;
}
```

这里创建了一个 HTML DOM `tbody` 对象，然后抓取了最近 10 条数据的键（最后的 `reverse()` 函数使最新的数据显示在表格的最上面一行）到一个数组，然后遍历这个数组，循环地创建表格行，并给这一行创建 3 个单元格，然后把这一行附加到前面创建的 `tbody` 上。

最后一步是把当前显示的 `tbody` 替换成新创建的，函数代码如下：

```
function updateHistoryTable(symbol) {
var tbody = makeHistoryTbody(fullHistory[symbol]);
var x = document.getElementById("history_" + symbol);
```

```
x.parentNode.replaceChild(tbody, x);
tbody.id = x.id;
}
```

最后一个文件，fx_client.history.html，用了一些 CSS 和一些响应式网页设计原则，所以页面很好看，而且在桌面和移动浏览器上都能合理地布局。图 4-1 至图 4-3 分别展示了页面在刚加载完，运行几秒之后和运行一段时间之后的样子。

| USD/JPY | EUR/USD | AUD/GBP |
|---|---|---|
| | | |

| USD/JPY | | |
|---|---|---|
| 时间戳 | 报价 | 问价 |

| EUR/USD | | |
|---|---|---|
| 时间戳 | 报价 | 问价 |

| AUD/GBP | | |
|---|---|---|
| 时间戳 | 报价 | 问价 |

图 4-1：fx_client.history.html，刚刚开始

seed=1389162952.3381

| USD/JPY | EUR/USD | AUD/GBP |
|---|---|---|
| 94.550 | 1.30096 | 1.43697 |

| USD/JPY | | |
|---|---|---|
| 时间戳 | 报价 | 问价 |
| 2014-01-08 06:35:54.892 | 94.550 | 94.560 |
| 2014-01-08 06:35:54.228 | 94.591 | 94.601 |

| EUR/USD | | |
|---|---|---|
| 时间戳 | 报价 | 问价 |
| 2014-01-08 06:35:56.018 | 1.30096 | 1.30106 |
| 2014-01-08 06:35:55.521 | 1.30036 | 1.30046 |
| 2014-01-08 06:35:55.192 | 1.30003 | 1.30013 |
| 2014-01-08 06:35:53.910 | 1.30140 | 1.30150 |
| 2014-01-08 06:35:53.426 | 1.30092 | 1.30102 |
| 2014-01-08 06:35:52.957 | 1.30006 | 1.30016 |
| 2014-01-08 06:35:52.692 | 1.30006 | 1.30016 |

| AUD/GBP | | |
|---|---|---|
| 时间戳 | 报价 | 问价 |
| 2014-01-08 06:35:56.958 | 1.43697 | 1.43717 |
| 2014-01-08 06:35:56.518 | 1.43609 | 1.43629 |
| 2014-01-08 06:35:54.483 | 1.43463 | 1.43483 |

图 4-2：fx_client.history.html，运行大约 5 秒后

seed=1389162952.3381

| USD/JPY | EUR/USD | AUD/GBP |
|---|---|---|
| 94.041 | 1.30388 | 1.44424 |

| USD/JPY | | |
|---|---|---|
| 时间戳 | 报价 | 问价 |
| 2014-01-08 06:36:10.626 | 94.041 | 94.051 |
| 2014-01-08 06:36:10.175 | 94.081 | 94.091 |
| 2014-01-08 06:36:07.158 | 94.223 | 94.233 |
| 2014-01-08 06:36:03.411 | 94.272 | 94.282 |
| 2014-01-08 06:36:00.887 | 94.428 | 94.438 |
| 2014-01-08 06:35:59.306 | 94.383 | 94.393 |
| 2014-01-08 06:35:58.604 | 94.440 | 94.450 |
| 2014-01-08 06:35:54.892 | 94.550 | 94.560 |
| 2014-01-08 06:35:54.228 | 94.591 | 94.601 |

| EUR/USD | | |
|---|---|---|
| 时间戳 | 报价 | 问价 |
| 2014-01-08 06:36:12.127 | 1.30388 | 1.30398 |
| 2014-01-08 06:36:11.560 | 1.30365 | 1.30375 |
| 2014-01-08 06:36:11.125 | 1.30368 | 1.30378 |
| 2014-01-08 06:36:09.311 | 1.30339 | 1.30349 |
| 2014-01-08 06:36:08.962 | 1.30301 | 1.30311 |
| 2014-01-08 06:36:08.208 | 1.30308 | 1.30318 |
| 2014-01-08 06:36:07.805 | 1.30240 | 1.30250 |
| 2014-01-08 06:36:07.444 | 1.30231 | 1.30241 |
| 2014-01-08 06:36:06.768 | 1.30125 | 1.30135 |
| 2014-01-08 06:36:06.092 | 1.30236 | 1.30246 |

| AUD/GBP | | |
|---|---|---|
| 时间戳 | 报价 | 问价 |
| 2014-01-08 06:36:11.867 | 1.44424 | 1.44444 |
| 2014-01-08 06:36:09.702 | 1.44615 | 1.44635 |
| 2014-01-08 06:36:08.689 | 1.44457 | 1.44477 |
| 2014-01-08 06:36:06.513 | 1.44312 | 1.44332 |
| 2014-01-08 06:36:05.667 | 1.44456 | 1.44476 |
| 2014-01-08 06:36:02.942 | 1.44221 | 1.44241 |
| 2014-01-08 06:36:01.479 | 1.43989 | 1.44009 |
| 2014-01-08 06:35:57.849 | 1.43680 | 1.43700 |
| 2014-01-08 06:35:56.958 | 1.43697 | 1.43717 |
| 2014-01-08 06:35:56.518 | 1.43609 | 1.43629 |

图 4-3：fx_client.history.html，运行一段时间后

我不打算去探究 CVS 和网页设计，那偏离了本书主题，来看一下其中的一个表格：

```
<table class="price-table">
    <caption>USD/JPY</caption>
    <thead>
        <tr>
            <th>Timestamp</th>
            <th>Bid</th>
            <th>Ask</th>
        </tr>
    </thead>
    <tbody id="history_USD/JPY"></tbody>
</table>
```

每个表格都有一个静态的标题栏和表头，给 `tbody` 设置一个 `id`，这样能找到它并且只替换表格的那部分。



所有支持 SSE 的浏览器都支持 `Object.keys`，所以 `makeHistoryTbody()` 中没问题。但实现向后兼容时，需要对 IE8 以及更早· 版本做兼容，后面第一次需要它时会介绍这种兼容。注意，这种兼容在效率上会慢一些：它是一个复杂度为 O(*n*) 的算法（*n* 是键的数量），而原生的 `Object.keys` 复杂度是 O(1)。

最后提醒一下，当你看到这个页面努力地忙着更新着自己时，别忘了我们只是故意让它只显示每个外汇对的最近 10 条问价 / 报价，但我们把所有接收到的问 / 报价都存储在内存中，你需要在功能性和客户端资源（这里是指可用的内存）之间做一个权衡。如果你确信并不需要全部数据，考虑一下定期裁剪 `fullHistory` 对象。

## 4.5　永久存储

前面一节介绍了如何存储所有接收到的数据，这开启了一个充满可能的世界：表格，实时图表，运用最新机器学习技术进行的客户端侧数据分析等。你可以依靠浏览器把握市场的脉搏，但你只要关了它，也就意味着所有已经下载的数据、计算结果，统统不见了。

让 HTML5 来救场吧！事实上，我们有选择，但是，文件系统（`FileSystem`）还没有被广泛支持，索引数据库（`IndexedDB`）也是如此（虽然有一种兼容方案可以让它的支持更多一点），所以我们选择了 Web 存储。这些新的 HTML5 存储 API 共同的限制是必须得到用户同意，并且存储空间是按域分配的，第 9 章会介绍域的确切定义，大概意思就是 http://example.com/ 上运行的应用，不能访问 http://other.site/ 上运行的应用创建的数据。

Web 存储更常见的叫法是 localStorage，允许存储名 / 值对数据，通常浏览器会给一个应用提供 5 MB 的存储空间，最赞的是几乎所有浏览器都支持 Web 存储：IE8 以上版本（有 IE6/IE7 的兼容方案）、Firefox 3.5 以上版本、Chrome、Safari 4 以上版本、Android 2.1 以上版本，以及 Opera 10.5 以上版本（参见 http://caniuse.com/namevalue-storage）。

Web 存储的缺点是它不能存储结构化的数据，只能是字符串，这意味着前文提到的历史记录数据必须用 `JSON.stringify()` 转化为字符串，这有点低效，如果数据量大，字符串和对象之间的转化还会有内存和 CPU 时间方面的开销。

使用 Web 存储的代码很简单，只需对现有代码进行两处修改。首先，要保存数据，把这一行插到 `processOneLine(s)` 中：

```
function processOneLine(s){
var d = JSON.parse(s);
...
else if(d.symbol){
    if(!fullHistory.hasOwnProperty(d.symbol))fullHistory[d.symbol] ={};
    ...
    updateHistoryTable(d.symbol);
    localStorage.fullHistory = JSON.stringify(fullHistory);
    }
}
```

是的，就这么简单。`JSON.stringify()` 把 `fullHistory` 对象转化成字符串，对 `localStorage.XXX` 赋值既可以创建一个 XXX 键，也可以替换它，也可以这样写：`localStorage.setItem("fullHistory", JSON.stringify(fullHistory));`。

注意这里在发生什么：每次有数据传过来，整个历史记录数据就被转化成字符串，替换原来的值。在我的测试里，Firefox 上运行了大约一个小时后，占用了 25% 的 CPU 资源（刚开始的时候是 4%），字符串大小达到了 500 KB，虽然这还不是很要命，但再运行几个小时就会了。

## 优化

有两种可能的优化，可以用外汇对把数据分割，存储历史记录的代码就会变成这样：

```
localStorage.setItem("fullHistory." + d.symbol,
    JSON.stringify( fullHistory [d.symbol]) );
```

我们有 3 个外汇对，这意味着分割之后每个字符串大小是原来的 1/3，这在性能上的收益是，如果原来发生故障的时间是 4 小时后，那现在是 12 小时后。把这个思路延伸一下，还可以把时间戳的一部分用在键名中。比如，可以把每个小时的数据归到一组。这确实增加了一点复杂度，因为现在需要额外的 `localStorage` 条目来记录数据的所属时段（替代方案是，localStorage API 提供了一种遍历**全部**存储数据的方式：`for(var ix = 0; ix < localStorage.length; ix++){var key = localStorage.key(i); … })`。

另一种优化方案是用 `setInterval` 每隔 30 秒保存一次数据，而不是每次获得数据的时候。这显著地降低了 CPU 的使用，但请注意，在运行了一段时间之后，CPU 的占用还是会达到 100%，只是不会一直 100%，而是每 30 秒爆发一次；另一件需要注意的事是，关闭浏览器时，在上一次保存之后传过来的数据会丢失。

代码需要做的另一个修改是使用永久存储的数据，在 connect() 的最上面加上下面几行：

```
function connect() {
if (localStorage.fullHistory) {
    fullHistory = JSON.parse(localStorage.fullHistory);
    updateHistoryTable("USD/JPY");
    updateHistoryTable("EUR/USD");
    updateHistoryTable("AUD/GBP");
    }
if (window.EventSource)startEventSource();
// 否则在这里处理向后兼容
}
```

这就是简单地把数据保存的过程反过来，用 JSON.parse 取代 JSON.stringify，也可以写成 fullHistory = JSON.parse(localStorage.getItem("fullHistory"));。

代码中其他 3 行是用之前保存的数据更新视图。

如果它不生效，检查一下你的浏览器设置，有些浏览器的安全策略设置是存放在 cookie 中的，所以你可能也需要允许存储 cookie。如果你点击浏览器的刷新按钮时有效，关闭所有浏览器窗口再打开却无效，检查一下浏览器的安全策略设置，看其中是否有一项是当浏览器会话结束时删除所有 cookie。



要删除数据很简单，只要调用 localStorage.removeItem('fullHistory');。如果你按补充内容“优化”中提到的方案实现了按小时分组数据，那你可以用前面那句代码删除最老的数据。

能存储多少数据？这取决于浏览器，但通常可以达到 5 MB（足够让我们的 FX 示例应用运行 11~12 小时）。当达到存储限制会怎么样？一般情况下会导致 setItem() 的调用失败，抛出一个可以捕获处理的 QUOTA_EXCEEDED_ERR 异常，传入 setItem() 的 key 原来的数据依然保留。Opera 浏览器会先弹出一个对话框，询问用户是否允许提供更多存储空间。在 Firefox 中，用户可以通过修改 dom.storage.default_quota（在 about:config 中）的值来调整存储空间大小（在 about:config 中）。

## 如何减少数据大小

一种思路是在存储之前压缩 JSON 字符串，可以在网上搜到 zip、gzip 和 LZW[2] 的 JavaScript 实现，JSON 很好压缩。

注 2：LZW（Lempel-Ziv-Welch）是 Abraham Lempel，Jacob Ziv 与 Terry Welch 提出的一种无损压缩算法。——译者注

> 你也可以尝试对数据进行总结提炼后再存储，而不是直接存储原始数据。这是有损压缩，而不是前文中推荐的无损压缩。比如，在金融类应用中，很普遍的做法是把原始的离散型数据转变成连续型数据，一个一分钟的连续数据只要存储开盘价格、收盘价格、最高价格、最低价格，以及这一分钟里的离散数据量；规划合理的话你所需的存储空间几乎保持不变。比如，你可以存储 10 分钟内的原始数据，2 小时内的一分钟连续数型数据，1 星期内的一小时连续型数据，以及几年内的一天连续型数据。

## 4.6 现在我们是历史学家

本章首先优化了第 3 章的代码结构，然后在此基础上介绍了如何存储历史数据，以及展示最新数据和一段历史数据；然后介绍了如何把这个和 Web 存储技术结合起来，这样客户端能有一个永久的数据缓存。现在已经完成了特性开发，下一章全部是关于如何使应用具备产品级品质。

# 第 5 章

# 走出象牙塔，打造产品级品质

前面两章创建了一个推送多个外汇对价格的后端，以及一个在支持 SSE 的浏览器上展示的前端。这个应用通过以下方式来改进：让它能在不支持 SSE 的旧版浏览器及移动浏览器上运行。但还有一个重要的地方需要改进，因为此时此刻我依然认为这只是个简陋的范例，它还没有达到产品级品质。



产品级品质是什么意思呢？它包括以下几点：当出现错误时系统能自动修复，能适应现实世界的各种限制（本章会介绍如何处理外汇市场周末歇业的场景），能够应对服务端发送错误数据的情况。

## 5.1　错误处理

在第 4 章中，我们给 error 事件绑定了事件处理程序，并将函数命名为 handleError。现在需要确定函数里面怎么写。本章的末尾会在客户端做自动重连功能，任何时候后端服务断开，客户端都会自动重连。即便后端不可用，也会一直尝试连接。不过，做这些和有没有 error 事件处理程序无关。error 事件处理程序只是提供信息的：只有程序员关心这一点，而用户并不关心。所以回调函数可以简单写成：

```
function handleError(e) {
console.log(e);
}
```

说“提供信息”有些言过其实。上文代码中的 e 对象没有消息，没有错误码。唯一有一点用处的是 target 字段。它是触发事件的 EventSource 对象，可以从中找到曾连接过的 URL，以及一个 readyState（完整的是 e.target.readyState）字段。如果 readyState 是 2

（或者 `e.target.CLOSED`），即意味着“关闭”，意味着 URL 有问题，并没有成功建立连接，如果 `readyState` 是 0（或者 `e.target.CONNECTING`），意思是“连接中”，意味着之前建立的连接已经关闭并且浏览器在尝试自动重连，如果 `readyState` 是 1（或者 `e.target.OPEN`），意味着已成功建立连接。

## 5.2 错误的JSON

如果服务端发送的字符串不是 JSON 格式的，或者格式有问题（少了一个逗号或换行符），浏览器可能会抛出异常。这会让一切都停止运作，这很糟糕。所以，只写一句 `var d = JSON.parse(s);` 是不够的，达到产品级品质要这样写：

```
try {
    var d = JSON.parse(s);
} catch(e){
    console.log("BAD JSON:" + s + "\n" + e);
    return;
}
```

## 5.3 长连接

有时我会一走几个星期甚至几个月都不给我妈妈打个电话，但她怎么知道我到底是没事要说，还是手机欠费了，还是被流星砸了躺在医院里呢？所以，偶尔我会给她发邮件，告诉她“账单都付清了，没被流星砸到。”或者更简单：“我还活着。”

在网络术语里，长连接是每 N 秒发送的一个数据包，或者在一个套接字失活 N 秒后，告诉套接字的另一头一切正常并且没有什么需要通信的。（有时会看到这个概念被称为心跳）。有些浏览器可能会断开连接，并在套接字失活一段时间后关闭重连。另外，代理服务器会在一个连接沉默后将其关闭。为防止这种情况发生，我们在应用中每 15 秒发一个长连接消息。为什么是 15 秒？ SSE 协议草案提到了这个数字。它可能比实际所需的频率要快一些，但换个角度讲，这个频率也还不至于成为系统的瓶颈。

所以，这样就确定了 N 的值。接下来要做的设计决策是：是每 15 秒发一次长连接消息，还是在沉默了 15 秒后发送。这不是很重要，你看在服务端怎么编写代码最简单就怎么做。

注意，长连接会影响 TCP/IP 带宽控制，尤其是它的慢启动重启机制。别担心，如果 15 秒，30 秒或者 90 秒的 SSE 长连接对网络负载有显著影响，可能是因为别的地方出了更大的问题（给初学者们提个问题，为什么有这么多没有发送任何实际数据的 SSE 连接呢？）配置服务器不使用慢启动可作为替代方案（这是个操作系统级别的设置）。

在移动设备上，关于长连接还有一些别的注意事项。比如，长连接可能会阻止应用进入休眠，因此会加快设备电量消耗。如果数据更新频率不高但可预测，考虑一下用 `setTimeout`

来获取数据，而不是用推送。

## 5.3.1 服务器端

长连接可以简单地通过发送 SSE 注释行来维持。怎么做呢？它只是一行以分号开始的代码。你可能记得 3.7 节中我们用 SSE 注释来输出一些问题诊断。举例如下：

```
echo ":\n\n";@flush();@ob_flush();
```

我们还可以发一个空数据信息：

```
echo "data:\n\n";@flush();@ob_flush();
```

发送 SSE 注释行和发送 SSE 数据行有什么区别呢？在服务端是 4 字节的区别，但在客户端那就是天壤之别了。后者触发了 `EventObject` 的消息事件处理程序，而前者没有[1]。我们想要后者，具体原因会在后面讨论客户端事件处理时谈到。

发送实际的数据包时，也包含一个时间戳（这个可用来检测服务端与客户端时间是不同步了还是高延迟）。因为消息是 JSON 格式的，所以指明这是长连接消息并不麻烦，代码如下：

```
sendData(array(
    "action" = > "keep-alive",
    "timestamp" = > gmdate("Y-m-d H:i:s")
    ));
```

在本书源码的 fx_server.keepalive.php 文件中找到使用这段代码的例子。因为本书的应用是持续不断地发送数据，永远不会有 15 秒的沉默，所以永远不会发送一个长连接。（长连接这个概念对这个应用基本上没什么意义。）但是为了能在前端测试，这里用了定期发送模式，在无限循环主体开始之前初始化 `$nextKeepalive = time() + 15;`，简单地说，这句代码的意思就是说，发送下一个长连接消息的时间是从现在开始的 15 秒以后，然后循环主体就在休眠之后开始执行，刚刚在一次休眠之后，代码如下所示：

```
while (true) {
    ...
    if (time() > $nextKeepalive) {
        sendData(array(
        "action" => "keep-alive",
        "timestamp" => gmdate("Y-m-d H:i:s")
        ));
    $nextKeepalive = time() + 15;
    }
$ix = mt_rand(0, count($symbols) - 1);
...
```

注 1：写作本书时，所有浏览器悄悄吃掉了 SSE 注释，所以你甚至在开发者工具中都看不到它们。

要把它修改为只在沉默期后发送，只需修改成在发送数据后运行 `$nextKeepalive = time() + 15;` 即可。

### 5.3.2 客户端

SSE 协议已经包含了一套重连机制。所以，我们并不是一定需要用长连接方案，只需发送一个 SSE 注释（见上一节所述）就足够保持 TCP/IP 套接字连接。如果套接字死掉了，浏览器会自动重连。之所以没有依赖这套重连机制，有两个原因，第一个原因是只有套接字死得优雅干脆利落时浏览器才会重连，就像动作电影里的群众演员一样。但是，有时候套接字死得像动作电影里的英雄一样。就像电影里那样，套接字停止工作之后，浏览器可能要在 30 秒、60 秒甚至 120 秒后才确定它死掉了。后端代码的 bug 会引起类似的问题。第二个原因很合理而且简单：代码要适用于向后兼容方案，这也是为什么长连接消息要像普通消息一样能用 JavaScript 处理。

首先需要定义两个 JavaScript 全局变量：

```
var keepaliveSecs = 20;
var keepaliveTimer = null;
```

第一个变量决定了灵敏度，一个合理的值应该能匹配服务端发送长连接消息的时间间隔。现在后端的间隔是 15 秒，考虑到网络延迟和其他可能在前后端的延迟，这里加了一点盈余，选择了 20 秒这个值。

`keepaliveTimer` 是 `setTimer()` 的回调函数，这个计时器会在初始化连接时创建。然后，每次有数据从服务端过来，就会销毁原来的计时器并创建一个新的。所以，只要有数据（不论是真正的数据还是长连接消息）定期传过来，计时器就会总是在触发之前被销毁。只有在 20 秒内没有数据传过来时，计时器才会触发。这说明某个地方出了问题，因为客户端本该每 15 秒收到一次长连接消息。

这部分代码如下所示：

```
function gotActivity() {
if (keepaliveTimer != null)clearTimeout(keepaliveTimer);
keepaliveTimer = setTimeout(connect, keepaliveSecs * 1000);
}
```

`setTimeout` 的第二个参数是以毫秒为单位的，因此 `keepaliveSecs` 乘以 1000。第一个参数是计时结束时要调用的函数，所以当没有接收到长连接时就会调用 `connect()`。还记得 `connect()` 函数吧，它以前是这样的：

```
function connect() {
if (window.EventSource)startEventSource();
// 否则在这里处理向后兼容
}
```

要让其开始运行，只需要加一行代码，如下所示：

```
function connect() {
gotActivity();
if (window.EventSource)startEventSource();
// 否则在这里处理向后兼容
}
```

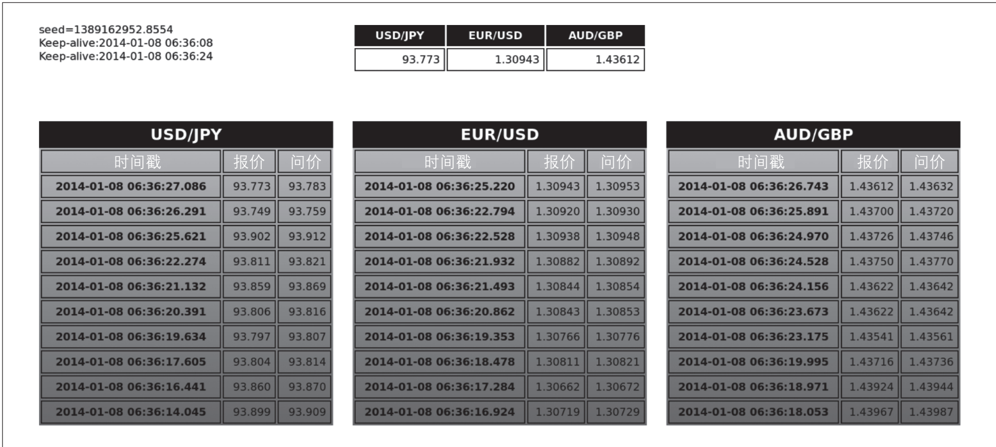
这很重要，因为如果没有它，当连接出错时（永远不会发任何数据）时就不会被发现。将 `gotActivity()` 放在 `connect()` 函数开始的地方，是为了防止 `startEventSource()` 抛出异常，并且后面的代码不会被执行到。

为什么没有在 `connect()` 中销毁旧的连接？这项工作留给了 `startEventSource()`（其实已经做了处理，参见 4.3 节）。在不同的向后兼容方案中，其销毁连接的方式是不同的。

最后，在 `processOneLine(s)` 的最上面，在 `gotActivity()` 上添加一个调用：

```
function processOneLine(s) {
gotActivity();
var d = JSON.parse(s);
...
```

这与它是否是一个长连接，是否定期发送数据，以及其他东西都没关系。程序执行到 `processOneLine(s)` 表明已接收到消息。接下来两章讨论的向后兼容方案中也会用到 `connect()` 和 `processOneLine(s)`，所以这段代码可以不作修改就能被它们用于支持长连接。运行一下 fx_client.keepalive.html 看看实际效果。图 5-1 显示了有两次长连接传过来的效果。




| USD/JPY | EUR/USD | AUD/GBP |
|---|---|---|
| 93.773 | 1.30943 | 1.43612 |

USD/JPY

| 时间戳 | 报价 | 问价 |
|---|---|---|
| 2014-01-08 06:36:27.086 | 93.773 | 93.783 |
| 2014-01-08 06:36:26.291 | 93.749 | 93.759 |
| 2014-01-08 06:36:25.621 | 93.902 | 93.912 |
| 2014-01-08 06:36:22.274 | 93.811 | 93.821 |
| 2014-01-08 06:36:21.132 | 93.859 | 93.869 |
| 2014-01-08 06:36:20.391 | 93.806 | 93.816 |
| 2014-01-08 06:36:19.634 | 93.797 | 93.807 |
| 2014-01-08 06:36:17.605 | 93.804 | 93.814 |
| 2014-01-08 06:36:16.441 | 93.860 | 93.870 |
| 2014-01-08 06:36:14.045 | 93.899 | 93.909 |

EUR/USD

| 时间戳 | 报价 | 问价 |
|---|---|---|
| 2014-01-08 06:36:25.220 | 1.30943 | 1.30953 |
| 2014-01-08 06:36:22.794 | 1.30920 | 1.30930 |
| 2014-01-08 06:36:22.528 | 1.30938 | 1.30948 |
| 2014-01-08 06:36:21.932 | 1.30882 | 1.30892 |
| 2014-01-08 06:36:21.493 | 1.30844 | 1.30854 |
| 2014-01-08 06:36:20.862 | 1.30843 | 1.30853 |
| 2014-01-08 06:36:19.353 | 1.30766 | 1.30776 |
| 2014-01-08 06:36:18.478 | 1.30811 | 1.30821 |
| 2014-01-08 06:36:17.284 | 1.30662 | 1.30672 |
| 2014-01-08 06:36:16.924 | 1.30719 | 1.30729 |

AUD/GBP

| 时间戳 | 报价 | 问价 |
|---|---|---|
| 2014-01-08 06:36:26.743 | 1.43612 | 1.43632 |
| 2014-01-08 06:36:25.891 | 1.43700 | 1.43720 |
| 2014-01-08 06:36:24.970 | 1.43726 | 1.43746 |
| 2014-01-08 06:36:24.528 | 1.43750 | 1.43770 |
| 2014-01-08 06:36:24.156 | 1.43622 | 1.43642 |
| 2014-01-08 06:36:23.673 | 1.43622 | 1.43642 |
| 2014-01-08 06:36:23.175 | 1.43541 | 1.43561 |
| 2014-01-08 06:36:19.995 | 1.43716 | 1.43736 |
| 2014-01-08 06:36:18.971 | 1.43924 | 1.43944 |
| 2014-01-08 06:36:18.053 | 1.43967 | 1.43987 |

图 5-1：运行大概 35 秒后的 fx_client.keepalive.html；fx_client.keepalive.html 收到两次长连接的效果

### 另一种做长连接的方法

现在的长连接方案是每次获得新数据时就销毁原来的计时器，然后再新建一个。有一种替代方案是只记录上一次数据的时间戳，然后可能需要一个每 4 秒跳动一次的计时器。每次计时器触发，就会检查一下距离上一次获取数据有多长时间。当超过 20 秒时，它会认为服务器已经挂了，这时就会开始重连。

这种方式有一些缺点。它需要再定义两个全局变量（`var keepaliveTimerSecs=4;`，`var lastTimestamp=null;`）；需要差不多两倍的代码量。另外，它还不及现方案精确；服务器出问题后，前端需要大概 20 到 24 秒才能识别到。前文所述的方案里，可以刚好在收到上一条数据的 20 秒后识别到。

这种方案一定是有优势的，对吧？是的，在每次收到数据时，更新一个时间戳比销毁然后重启一个计时器要快。这个额外的 CPU 负载有些无足轻重，只有当陷入到非常迅猛的数据爆发，并且已经忙得不可开交时，这个额外的 CPU 负载才会出现。谢谢你问这个问题。

本章的第一个草案中，是在正文里介绍这种方案的，而更简版本则是在附注里介绍的。可是，我有些怀疑，所以就去测试了一下两者的差异。在 Chrome（基于 WebKit/PhantomJS）中，销毁重建计时器方案的耗时是更新时间戳方案耗时的 14~17 倍。在 Firefox 中，这个差异更大，大约相差 250~350 倍！所以我的直觉是对的，甚感欣慰！然后我往后退了一步，进行了一次微优化。以每秒接收 100 条消息为例，这可以算是很快的数据更新频率了。测试表明 100 次销毁 – 重建计时器耗时大概 6 毫秒，所以每秒 100 条消息相当于占用了 0.6% 的 CPU 开销。

结论：销毁 – 重建计时器的长连接处理方案永远不会成为性能瓶颈。

## 5.3.3　SSE重试

SSE 有内置的保持连接机制。它是怎样工作的？代码如何和它进行交互？ SSE 内置的重连机制是在套接字层级上运作的，如果服务器关闭了套接字，浏览器会执行如下几步。

- 设置 `readyState`（`EventSource` 对象的一个属性）值为 `EventSource.CONNECTING`。
- 调用 `error` 事件处理程序（参见 5.1 节）。
- 等待 `retry` 延时时间，然后重新连接。

重试等待时长由什么决定？默认是由浏览器决定的[2]，大约 3~5 秒。这意味着如果连接因为套接字关闭而断开，在长连接代码检测到之前，内置的重连就已经发生了。所以这没什么冲突，SSE 重连会处理一切，长连接重连代码永远不会用上。

注 2：写作本书时，在 Chrome 和 Safari 中是 3 秒（参见 WebKit 或 Blink 源码的 core/page/EventSource.cpp 文件），在 Firefox 中是 5 秒（参见 Mozilla 源码的 content/base/src/EventSource.cpp 文件）。

既然 SSE 有内置的重连机制，为什么还要费劲写一套长连接机制？问得好。首先，在接下来两章介绍的向后兼容方案需要它，并且即便我们控制了生态系统，而且知道所有的浏览器都能支持 `EventSource`，仍然需要这段代码。SSE 内置重连只能处理套接字关闭这一种引发错误的情况，而其他导致连接出错的情况还有很多，可能套接字死掉了却没有被检测到，可能后端脚本发生崩溃并且没有正常地关闭套接字，可能进入了死循环，可能浏览器或服务器出现了 bug。幸好，我们这个显式的长连接机制能处理所有这些情况，但最重要的是它能处理服务端离线的情况。当网络服务无法访问，或者返回 404，或者跨域配置有问题，SSE 把 `readyState` 值设为 `EventSource.CLOSED`，并不再重试，而显式长连接机制每 20 秒会重试一次。

回到 SSE 内置重连，默认等待 3~5 秒确实很短。如果服务器可能会经常关闭连接并且不需要进行频繁的重连，可以把重连时间设大一点。相反，如果想要更短一点，以减少出错后的故障时间，可以设置一个更小的值[3]，可以通过在 SSE 消息中添加下面这一行来修改：

```
retry: 10000
```

它的单位是毫秒，所以 10 000 意味着 10 秒。建议不要将这个值设成大于发送长连接消息的时间间隔。比如，如果把这个重连时间设置为 20 000，那发送长连接的时间间隔建议设置为 25~30 秒（但也不要比重连时间大太多，不然浏览器、代理服务器等中转媒介、负载均衡器等会把沉默当成丢失连接）。请记住，如果增大服务端的重连时间间隔，务必也要增大 JavaScript 中的 `keepaliveSecs`。

也许我们可以从 SSE 的内置重连获得灵感，来实现一套我们自己的协议？这样服务端规定一个发送长连接的频率，然后自动调整与之匹配的 `keepaliveSecs` 的值。这是个非常好的想法，当服务端超载，动态地告知客户端调整一下重连时间。事实上，如果把发送长连接消息的时间间隔设置太长，浏览器（或者中转媒介）会认为出错了，并关闭套接字，尝试重连，这样整体上会造成更多负载。所以只能在 15~40 秒这个范围内调整重连时间。这个收益不高，不值得为此把它弄得更为复杂。

本书的源码里有一个 fx_server.retry.php 文件，它在脚本最上面加了一行代码，如下所示：

```
header("Content-Type: text/event-stream");
echo "retry: 10000\n\n";@flush();@ob_flush();
...
```

怎么测试呢？这个脚本包含了一段自毁语句！在无限循环的顶部，我添加了如下代码：

```
while(true){
    if(time() % 20) == 0)break;
    ...
```

注 3：Firefox 强制要求最小的重连时间是 0.5 秒。

在每一分钟的开始，以及开始后的 20 秒和 40 秒之后，脚本会悄悄地退出。这个退出干净优雅，所以浏览器能立刻识别到。

### 销毁的方式

另一种测试 SSE 重连的方式是关闭服务软件。比如在 Ubuntu 上使用 Apache，只输入 `sudo service apache2 restart` 就可以了。正如前文所述的中断无限循环的方式那样，这也是一种干净的销毁方式：浏览器几乎能立即检测到套接字已经销毁。

顺便说一下，`sudo service apache2 graceful` 偶尔也正好是你不想要的——它会重启所有闲置的 Apache 实例，但 SSE 进程并没有闲置，所以不会关闭 SSE 套接字。

脏的销毁方式是怎样的呢？

如果服务器和客户端在不同的机器上运行，可以拔掉它们之间的网线。类似地，也可以关闭服务器的网络接口。浏览器不能识别出套接字已经断开连接，而长连接方案就能处理这种情况。

在使用 Apache 时，还有一种方式可以用于找出为请求提供服务的 Apache 进程 pid，然后执行 `sudo kill -s STOP 12345`。（这里的 12345 是进程 pid。）这种方式的效果类似拔掉网线，浏览器不能检测出问题，而长连接处理方案可以。STOP 标记方式进入休眠状态，用 `sudo kill -s CONT 12345` 可以将其重启（类似把网线插回去）。

为什么在前面两段所述的场景中，浏览器不能检测出问题呢？想象一下把网线拔出片刻再插回去，或者通过手机或 Wifi 信号上网时，短暂地走出信号范围又回来。TCP/IP 有针对这类短暂中断的处理设计。客户端不能区分这是沉默、临时问题还是致命问题，这就是需要一个长连接处理机制的原因。

可以拿一个客户端脚本连接到 fx_server.retry.php 试一下效果。（fx_client.retry.html 是为此而设计的，唯一变动的地方是它连接的 URL。注意它有一个长连接逻辑，所以你可以体验一下浏览器的长连接机制和我们自己的长连接机制之间的交互。）通过 `retry:10000`，可以看到服务器活跃 1~20 秒，然后沉默 10 秒。如果打开 JavaScript 控制台，可以看到出现一个错误：当浏览器检测到套接字消失时就会报错。然后可以看见交替地活跃 10 秒（新建连接的消息会打印到屏幕），沉默 10 秒。试一下注释掉 fx_server.retry.php 的重试代码，在 Firefox 中（`retry` 的默认值是 5 秒），可以看到 15 秒的活跃和 5 秒的沉默交替进行。现在试一下把 fx_server.retry.php 的 `retry` 值改为 500（也就是半秒），就可以在控制台日志中看见报错信息，但几乎没有服务中断。

最后，尝试把 `retry` 设置为 21 000。这比我们的 20 秒重连检测时间要长，所以是长连接机制处理重连，而不是浏览器的 SSE 机制。现在有趣的事情发生了，在每分钟的 0 秒、20 秒和 40 秒时关闭连接，这刚好匹配自毁时间，然后就没有数据传过来了！这真是有趣！确切地说，这只是自毁时间和长连接超时时间之间的偶然互动。尝试修改自毁时间或

JavaScript 中的 `keepaliveSecs` 来感受一下。或者更好一点，保持 `retry` 比 `keepaliveSecs` 小，并且把自毁代码放入代码中。嗯，不要着急，自毁也是下一节的主题。

## 5.4 添加定期的关闭/重连

在现实世界的外汇市场，周末没有数据可推送[4]。所有套接字都开着，但所有发送的数据是长连接消息。尤其在如今这个云服务时刻都在改变运算能力的时代，保持套接字连接却只发送长连接消息，着实是一种浪费。所以我们希望服务器能广播消息说："乡亲们，周一见！"。

可以在后端加上如下代码[5]：

```
$when = strtotime("next Sunday 17:00 EST");
$until = date("Y-m-d H:i:s T", $when);
$untilSecs = time() - $when;
sendData( array(
    "action" => "scheduled_shutdown",
    "until" => $until,
    "until_secs" => $untilSecs
    ));
```

这段代码把应该重连的时间戳发送给了客户端，这个值在 `until` 字段中。我们还发送了一个 Unix 时间戳 `until_secs` 字段，这方便客户端的处理（这也意味着客户端不需要担心时区不同的问题，或者服务器时钟偏慢的问题：服务器说 100 000 秒后再回来，这就是我们要做的）。

这里选择了纽约冬季时间的周日下午 5 点，这也是传统的外汇交易开始时间。计算 `$until` 的方式有一点粗糙，如果已经是周日，那么"下周日"（next Sunday）就会导致可怕的错误，其次，纽约会在夏天从 EST(UTC-05) 切换到 EDT(UTF-04)。或者直白点说，我们想要从三月的第二个周末到十一月的第一个周末都用 EDT。PHP 能自动做这些计算，但那超出了本书的范围。在真实应用中也需要考虑公共假日，所以应该考虑从一个数据库中获取所有的关闭和重连时间，而不是通过计算来获得。

事实上，这里会做一些类似的事情（参见 fx_server.shutdown.php），主循环现在会从磁盘上找一个叫 shutdown.txt 的文件，可以找到 `strtotime` 能解析的日期戳。

---

注 4：我们本该在前面章节介绍的模拟服务端中实现这个功能：定期查看时间，然后在纽约时间的周五下午 5 点进入一段时长 48*3600 秒的休眠。但是我让它以 24/7 的方式工作，是因为你可能需要在周末时调试示例脚本，这也太现实主义了！

注 5：注意：这里假设脚本是运行在 UTC(GMT) 时区，如果服务器不是在 UTC 时区，那就在 PHP 脚本的顶部加上 `date_default_timezone_set('UTC');`。或者也可以把当地时间戳传给 `strtotime`（但这会给客户端带来更多的工作量）。



这是本书第一次用到 `strtotime`，如果你不熟悉的话，参见附录 C.4 节。

然后它会把关闭的时间戳发给客户端。这段代码被添加到了主无限循环的开始部分：

```
$s = @file_get_contents("shutdown.txt");
if($s){
    $when = strtotime($s);
    $untilSecs = $when - time();
    if($when > 0 && $untilSecs > 0){
        $until = date("Y-m-d H:i:s T",$when);
        sendData(array(
            "action" => "scheduled_shutdown",
            "until" => $until,
            "until_secs" => $untilSecs
            ));
    break;
    }
}
```

第一行用 `@` 来抑制错误消息。实际上，这里先检查文件是否存在，然后再加载。如果文件不存在，`$s` 会是一个非真的值。其他代码基本上是前文中出现过的示例，加了一点对时间戳的错误检测（因为时间戳是从一个文件中读取的，可能会有一些意料之外的状况）。

因为这里是用夏季时间，所以在 EDT 时区周五下午 5 点时，需要创建一个文件，这个文件的内容是："next Sunday 17:00 EDT（EDT 时区的下周日 17:00）"。务必确保在周六午夜时删除这个文件。如果确实不希望客户端在周日白天连接，可以在周日的 17:00 之前把这个文件的内容替换为"17:00 EDT"（EDT 时区 17:00）。

现在来看一下前端怎么处理。有两个任务：一个是如何识别收到一条包含关闭时间的消息，另一个是如何执行关闭。首先，在主循环结尾添加下面这段代码：

```
...
else if(d.action=="scheduled_shutdown"){
    document.getElementById("msg").innerHTML +=
        "Scheduled shutdown from now. Come back at :" +
        d.until + "(in " + d.until_secs + " secs)<br/>";
    temporarilyDisconnect(d.until_secs);
    }
```

`temporarilyDisconnect()` 函数的第一稿如下所示：

```
function temporarilyDisconnect(secs){
var millisecs = secs * 1000;
if(keepaliveTimer){
    clearTimeout(keepaliveTimer);
```

```
        keepaliveTimer = null;
        }
    if(es){
        es.close();
        es = null;
        }
    setTimeout(connect, millisecs);
    }
```

停止长连接计时器（不想在打算休眠的那段时间触发它），关闭 SSE 连接（向后兼容方案中的连接也需要在这里关闭），然后就在被告知的时刻调用 `connect()`。

我说的是“第一稿”，所以你知道这里会有些问题……但它真的运作得很好。测试一下：在浏览器中打开 fx_client.shutdown.html，然后在服务器中该文件所在的目录下创建一个 shutdown.txt 文件，在这个文件里写一个大概 30 秒后的时间戳，推荐用 24 小时制，并且显式地指明时区。比如，如果是伦敦的夏天，并且当前时间是下午 3:30:00，那就写“15:30:30 BST”（回想一下 `strotime()` 的工作原理，如果不指定日期，则默认为当天）。这个时间会被转化为 GMT 格式，所以在浏览器中会出现一条消息：“Scheduled shutdown from now. Come back at 2014-02-28 14:30:30 UTC(in 29 secs)[6]”。等待 29 秒后它又回来了，就像“见证奇迹的时刻”。

它运行得非常完美，到底有什么问题？给你个提示：它运行得非常完美并且精确地在约定的时间调用 `connect()`。发现这里面潜藏的危险了吗？回到外汇市场，想象一下你拥有 2000 个客户，设想一下在纽约时间周日 17:00:00 会发生什么。他们全部在同一瞬间试图连接，然后流量会变得非常惊人。

怎样避免这种情况呢？观察发现，让一些客户早一点连接其实真的没什么关系。那不如加上下面加粗的那两句代码：

```
function temporarilyDisconnect(secs){
    var millisecs = secs * 1000;
    millisecs -= Math.random() * 60000;
    if(millisecs < 0)return;
    if(keepaliveTimer){
        clearTimeout(keepaliveTimer);
        keepaliveTimer = null;
    }
    if(es){
        es.close();
        es = null;
        }
    setTimeout(connect, millisecs);
    }
```

在浏览器中打开 fx_client.jitter.html 看一下效果。它将客户端的连接尝试随机地均摊在约定

注 6：现在开始定期关闭，将在 UTC 时间 2014-02-28 14:30:30（29 秒后）回来。——译者注

连接时间的前 60 内，第二行的意思是，如果根本不需要休眠，那么甚至都不要重连。顺便说一下，应该至少在重连时间的 60 秒前删除 shutdown.txt，不然那些提前连接的客户端又被告知要关闭了。

## 5.5 发送Last-Event-ID

当丢失连接后重连，无论如何，重连后会获取到最新的数据。这很好，但对任何正常的有效数据来说，这意味着会有一段数据的缺失。在第 4 章中，我们费劲周折地保存所有下载的历史数据，但如果不能精确和完整地保持，那就没那么有价值了。

幸好，SSE 协议的设计者考虑到了这一点。在建立连接时，会发送一个 `Last-Event-ID` HTTP 请求头，它指定了推送数据应该从哪里开始。这个 ID 是一个字符串，并非必须是数字。

好消息是，在使用 XMLHttpRequest 和 ActiveXObject 的向后兼容方案中，可以用 `setRequestHeader()` 函数来模拟这种行为。坏消息是，在使用 `EventSource` 时不能手动地指定一个值。所以用 SSE 时只能用服务器之前发送的值，那意味着在一个全新的连接中完全不能指定它的值。`EventSource` 对象没有 `setRequestHeader()` 函数（至少现在还没有）。这是（少有的）向后兼容方案比 SSE 好的情况。



如果你认为限制发送 `Last-Event-ID` 请求头是出于安全考虑，这样服务器就能阻止访问旧数据。我想指出的是，你可以用任何主流语言的客户端 HTTP 库来解决这个问题，所谓的安全是错觉。

想象一下长连接机制触发重连的场景，或者正在用 HTML5 `LocalStorage` 对象保存历史数据时，用户刷新页面（因此可以知道他接收到的最后一个数据事件，包括它的 ID）。在这些场景中，需要在 URL 中发送 ID。所以，服务端需要同时查看 URL 和 `Last-Event-ID` 请求头。请求头应该总是优先的（因为那意味着这是一个 `EventSource` 的自动重连，意味着 URL 中的 ID 已经过期了）。

接下来要考虑的是，一个指定的 SSE 连接只有一个 ID。如果通过同一个连接推送不同的数据（比如，不同的外汇汇率），该怎么做呢？有一种简单的方法和一种复杂的方法。复杂的方法会在下一节介绍。这里会用到的简单方法是，使用当前时间，具体点说就是服务器的当前时间，即从 1970 年到现在的毫秒数，使用这个毫秒数是因为，这是 JavaScript 内置的格式，不需要转化。服务端该怎么做呢？只需在每条数据行之前，加一个包含了这个时间的 id 字段，这样客户端会收到一系列这样的数据，如下所示：

```
id:1387946750885
data:{"symbol":"USD/JPY","timestamp":"2013-12-25 13:45:51",↵
    "rows":[{"id":1387946750112,"timestamp":"2013-12-25 13:45:50.112",↵
    "value":98.995},{"id":1387946750885,"timestamp":"2013-12-25 13:45:50.885",↵
```

```
    "value":98.980}]}

id:1387946751610
data:{"symbol":"USD/JPY","timestamp":"2013-12-25 13:45:51",↲
    "rows":[{"id":1387946751610,"timestamp":"2013-12-25 13:45:51.610",↲
    "value":98.985}]}
```



id: 行也可以放在 data: 行的后面，只要它们都在标示 SSE 消息结束的空行前面就行了。

眼尖的读者会注意到 data: 行的数据格式和我们现在用的不一样。rows 中的每一项除了有时间戳外，还有一个 id 字段。这样做是因为在 SSE 中无法得到 id 项（有意思的是，在向后兼容方案中可以）。注意，在进行 JSON 编码后，id 是一个整数，而不是字符串。

我们从 FXPair 类中的修改开始。相对于 fxpair.structured.php，fxpair.id.php 中的 generate() 函数只修改了两行代码。新版的 generate() 如下所示，加粗部分是添加的代码：

```
public function generate($t){
$bid = $this->bid;
$bid += $this->spread * 100 *
    sin( (360 / $this->long_cycle) *
    (deg2rad($t % $this->long_cycle)) );
$bid += $this->spread * 30 *
    sin( (360 / $this->short_cycle) *
    (deg2rad($t % $this->short_cycle)) );
$bid += (mt_rand(-1000,1000)/1000.0) * 10 * $this->spread;
$ask = $bid + $this->spread;

$ms = (int)($t * 1000);
$ts = gmdate("Y-m-d H:i:s",$t).sprintf(".%03d",$ms % 1000);
return array(
    "symbol" => $this->symbol,
    "timestamp" => $ts,
    "rows" => array(
        array(
            "id" => $ms,
            "timestamp" => $ts,
            "bid" => number_format($bid, $this->decimal_places),
            "ask" => number_format($ask, $this->decimal_places),
            )
        )
    );
}
```

$t 是从 1970 年到现在的秒数，带了小数部分。要获得毫秒数 $ms，用 $t 乘以 1000（因为 $t 是精确到毫秒的，用 (int) 截去微秒部分）。然后把这个数字放在 "id"=>$ms 这一行发送给客户端。

要在 fx_server.id.php 中将 `id` 返回给 SSE 客户端，需要在 fx_server.shutdown.php 的基础上加两行代码。在文件最上面加上 `include_once("fxpair.id.php");`，引入新的 FXPair 类。然后在 `sendData()` 下面，添加另一个辅助函数：

```
function sendIdAndData($data){
$id = $data["rows"][0]["id"];
echo "id:".json_encode($id)."\n";
sendData($data);
}
```

它输出 `id:` 行，然后通过 `sendData()` 来输出 `data:` 行并刷新输出缓冲。



在刚刚介绍的 `sendIdAndData()` 中，`echo "id:".json_encode($id)."\n";` 这一行可以等同于 `echo "id:$id\n";`，因为 `$id` 是一个整数，不需要特别的 JSON 编码，修改一下会发现应用的表现是一样的。这里显式地使用 `json_encode()`，这样即便 `$id` 是一个字符串甚至更复杂的数据结构（见接下来一节）也可以运行。

接下来，在主循环的中段将下面这行代码：

```
sendData( $symbols[$ix]->generate($t) );
```

改为：

```
sendIdAndData( $symbols[$ix]->generate($t) );
```

要在浏览器中使用它，只需修改前端文件中连接的 URL，不需要做其他事情，因为 SSE 对 id: 的使用是由浏览器在后台处理的。



现在数据项中有一个整数类型的 ID，可以回到前面的历史数据存储，把它用作键，取代原来使用的时间戳字符串。这意味着键占 8 字节，而使用字符串时占用 24 字节。这可能意味着查找会更快些，但这里有个问题：我们也在接口中使用了这个时间戳字符串。所以我们或许仍然需要将其存储（需要更多的内存），或者需要用 JavaScript 的 `Date` 函数从毫秒值获得（虽然这可以更灵活地使用不同的日期格式，但需要更多的 CPU 资源）。我选择保持原样。

需要注意的是，ID 是（最新）数据的时间戳，*不是*当前时间。这在前面的代码里介绍了（我不指望你知道 ID 数字，但最后 3 位是秒数的小数部分，倒数第 4 位是秒数的最后一位）。当然，最新一条数据的时间戳和当前时间很接近，但考虑到发送给客户端的数据可能是通过一系列服务器串行地传过来的，延迟会因此累加。我们需要知道*数据*的 ID，因为当重连时，我们用那个 ID 来告诉服务器我们所看见的最后一条*数据*，以便它能刚好从下一条数据开始重发。

## 5.6 多路数据ID

如果数据不是按时间索引的怎么办？比如数据是来自一个使用自增式主键的 SQL 数据库会怎样？使用 `Last-Event-ID` 请求头中的时间戳，则需要通过时间戳查找，这种查找会慢，或者需要在数据库中新增一个索引列（这会降低数据库写入的速度）。这里真正需要的 `Last-Event-ID` 的值是所用主键的最后一个值。

但如果是轮询多个数据库表会怎样？比如，在聊天应用或社交网站中，会推送各种类型的消息：聊天消息、聊天请求、好友登录、好友退出、新好友请求等。需要通过 Last-Event-ID 来传送在每个数据库表中都有的最后一条数据的 ID。

听起来很难，对吧？但我有好消息。服务器通过 `id:`，客户端通过 `Last-Event-ID` 发送的这个 ID，可以是任意字符的字符串（确切点说，任何除 LR 或 CR 之外的 Unicode 字符）。既然在 `data:` 字段的数据使用了 JSON 格式，何不在 `id:` 字段也使用 JSON 呢？如下所示：

```
id:{"chatmessages":18304,"chatrequests":1048,"friendevents":8202}
```

这样做是有必要的，在金融行业以不同的价格销售延时数据是很普遍的。比如说，实时的股票市场数据能卖到很贵，但是雅虎和谷歌可以免费地提供延时 20 分钟的数据。如果只购买两个外汇对的实时数据，而其他的买延时数据，那么 `lastId` 变量就会一直是从现在往前的 20 分钟。要保证无论何时需要重连时，都不会使某些外汇对的 `lastId` 出错，做法如下：

```
id:{"live":1234123412,"delayed":1234123018}
```

在 id: 字段使用 JSON 对象，还有一件事需要注意：如果达到 100 多字节，就不能用 GET 请求了，需要用 Cookie。（为什么不用 HTTP POST ？参见 9.3 节。）如果达到了几千字节，通过 HTTP 请求头发送时会有问题（记住是 SSE 发送这个请求头，我们不能控制它）。特别是当请求头的总大小（请求行，所有的请求头，包括 user agent 和所有的 Cookie）超过 8 KB 时，大部分网络服务软件会报错（返回 413 状态码）。



旧版 nginx 的限制是 4 KB，但现在默认是 8 KB，并且可以配置，参见 http://wiki.nginx.org/HttpCoreMod ule#large_client_header_buffers。Apache 也可以做类似配置，参见 http://httpd.apache.org/docs/2.2/mod/core.html#limitrequestfieldsize。

所以，如果 `id:` 字段超过了 100 字节，想一下是否有更好的方式。比如，能否把用户会话的每一个数据的位置都保存在服务端，然后通过 Cookie 来引用这个会话？

## 5.7 使用Last-Event-ID

回到服务端脚本，如何运用 Last-Event-ID 数据头？当使用 PHP+Apache 时，浏览器发送的请求头会发生如下变化。

(1) 全部变成了大写。
(2) 加上了 HTTP_ 前缀。
(3) 被置于 $_SERVER 变量中。

可以在 fx_server.id.php 中看到如下代码：

```
if(array_key_exists("HTTP_LAST_EVENT_ID", $_SERVER)){
    $lastId = $_SERVER["HTTP_LAST_EVENT_ID"];
    }
elseif(array_key_exists("lastId", $_POST)){
    $lastId = $_POST["lastId"];
    }
elseif(array_key_exists("lastId", $_GET)){
    $lastId = $_GET["lastId"];
    }
else $lastId = null;
```

（下一节会介绍为什么使用了 $_POST 和 $_GET。）在通过 HTTP 请求获得了用于查找的 seed 之后，先通过下面这段代码给 $t 赋值，然后用 $t 来设置随机种子：

```
if($lastId)$t = $lastId / 1000.0;
```

换句话说，因为这只是个测试应用，基本上就把 Last-Event-ID 当做 seed 来用。这足够用来测试及理解 Last-Event-ID 的工作原理。在真实应用中，这里是要请求历史数据的地方，然后发送一个丢失数据的补丁。



安全提示！ lastId 是纯粹的用户输入，理论上会包含任何东西，绝对不要假设它只会包含前端 JavaScript 代码会赋予的值，黑客可以放任何他想放的东西在里面。

前面的代码是安全的，但这里的安全校验很巧妙，预期 $lastId 是数字类型的，当除以 1000.0 时，如果它不是数字类型，PHP 会先隐式地将其转化为数字。如果黑客将 lastId 的值设置为 {"hello":"tell me your password"}，在除以 1000.0 之前，lastId 的值会被转化为 0，这样 $t 的值就成了 January 1, 1970，黑客所能做的最坏的事情也就是使 $t 成为一个很久以前或以后的日期。

当 lastId 是其他类型的值时，那就需要做更多的工作来处理它并弄清潜在的风险并处理。本书不是关于网络安全的，所以建议看一下与你正在使用的后端语言相关的安全处理技术。

如何测试？与前文介绍过的测试 `retry:` 请求头一样（参见 5.3.3 节）。所以，关闭连接（打开 JavaScript 调试窗口，可以看到浏览器能检测出 SSE 套接字消失），几秒之后浏览器重新连接并且接着上一次的外汇对。



你会发现数据的开始时间滞后于当前时间！这只是因为是假数据，如果你有强迫症，实在受不了，可以去看看心理医生。

### 用 Node.js 获取 `Last-Event-ID`

这一节的代码完全基于 PHP 特性，如果用 Node.js 写会怎样？下面这段代码基于第 2 章介绍过的 basic_sse_node_servers.js 中的代码进行了修改：

```
var url = require("url");
...
http.createServer(function (request, response) {
    var urlParts = url.parse(request.url, true);
    if (urlParts.pathname != "/sse") {
        ...
        }
    var lastId = null;
    if (request.headers["last-event-id"]) {
        lastId = request.headers["last-event-id"];

        }
    else if (urlParts.query["lastId"])lastId = urlParts.query["lastId"];
    console.log("Last-Event-ID:" + lastId);

    //SSE 数据从这里输出

    }).listen(port);
```

（可以在本书源码的 basic_sse_node_server.headers.js 文件找到这段代码。）

HTTP 请求头在 `request.headers` 中，优雅简单，只需注意，它们全部转化成小写的了。

在函数的最上面，解析 `request.url`，然后就可以在 `urlParts.query` 中获得 GET 数据。

这里没有介绍怎么获取 POST 数据，那会复杂一点，虽然才多了 6 行左右的代码[7]。但真正复杂的地方在于解析 POST 数据是异步的。因此代码需要重构来使用回调函数，在附注内容里阐述这个有些过于复杂了。

注 7：关于这个主题的讨论可以参见 http://stackoverflow.com/q/4295782/841830。

## 5.8 在重连时发送ID

上一节介绍了如何从 Last-Event-ID 中获取 $lastId，也包含了从 POST 和 GET 数据的 lastId 字段中查找。这样我们可以在新建连接时指定 ID，而不依赖于底层的 SSE 协议。为什么需要这样做？因为 EventSource 现在还不能发送 HTTP 请求头？不，那为什么还需要？因为在下面两种情况下需要它。

- 当长连接触发时，是 JavaScript 来做重连，而不是浏览器实现的 SSE 机制。
- 当浏览器重载页面，可以从 Cookie 或 LocalStorage 对象中获取最后的 ID。

注意代码的处理顺序：先到先得。如果请求头中有，就用请求头中的，否则，就从 POST 数据中查找（事实上，这是针对后面两章介绍的向后兼容方案的，原生的 SSE 不支持 POST 数据）。如果既没有 Last-Event-ID 请求头又没有在 POST 数据中找到 lastId，那就只能从 URL 中查找 lastId。这很重要，因为当 SSEL 发送 Last-Event-ID 进行重连时，它将使用相同的 URL。如果把 GET 数据的优先级放在 Last-Event-ID 请求头之前，那会用到一个旧的 ID 而不是最新的。

客户端需要做哪些修改呢，先定义一个全局变量：

```
var lastId = null;
```

如果在 LocalStorage 对象中有一个永久值，可以用它来初始化 lastId。

只需将 lastId 附加到 SSE 的 URL 中，而不需要考虑其他兼容方案（其他方案可以使用 HTTP 请求头的方式）。所以只需修改 startEventSource()，而不是 connect()，现在代码如下所示：

```
function startEventSource() {
if (es)es.close();
es = new EventSource(url);
es.addEventListener("message",
    function (e) {
    processOneLine(e.data);
    }, false);
es.addEventListener("error", handleError, false);
}
```

修改之后是这样（粗体是修改的部分）：

```
function startEventSource() {
    if (es)es.close();
    var u = url;
    if (lastId)u += "lastId="
        + encodeURIComponent(lastId) + "&";
    es = new EventSource(u);
    es.addEventListener("message",
```

```
        function (e) {
            processOneLine(e.data);
        }, false);
        es.addEventListener("error", handleError, false);
    }
```

最后一步得益于在第 4 章中做的重构（给每一项数据添加 id 字段），在 processOneLine(s) 函数中，现在是这样的：

```
    for (var ix in d.rows) {
        var r = d.rows[ix];
        x.innerHTML = d.rows[ix].bid;
        full_history[d.symbol][r.timestamp] = [r.bid, r.ask];
        }
```

现在在循环的最后加一行，这样全局变量 lastId 总是最新的 ID。

```
    for (var ix in d.rows) {
        var r = d.rows[ix];
        x.innerHTML = d.rows[ix].bid;
        full_history[d.symbol][r.timestamp] = [r.bid, r.ask];
        lastId = r.id;
        }
```

同样，甚至在浏览器关闭后使用 Web Storage 来保存数据

## ID 和多路数据源

记住，正文中介绍的方案（一个全局的 lastId）只适用于所有的外汇对（也就是多路数据推送）共用一套 ID 系统的情形。在我们的场景中，所有的外汇对用 id 代表数据时间（从 1970 年到现在的毫秒数）。

但即便用时间戳作为唯一 ID，也仍然需要注意。如果系统是广播来自两个或更多交易所的数据，这两个数据可能不是很同步，或者其中一个有临时的延迟。举个例子，来自纽约证券交易所的最新数据是 14:30:27.450 的，而纳斯达克的最新数据还是 14:30:22.120 的，有 5 秒的数据延迟，这时连接断开了。重连时，如果以 14:30:27.450 作为最后数据的时间，就会丢失纳斯达克 5 秒的数据；反之，如果用 14:30:22.120，就会有 5 秒的纽约证券交易所的重复数据。

所以处理两个数据源时，需要分别维护各自的最后 ID（参见 5.6 节）。

要测试这个，需要强制长连接计时器超时，意味着脚本需要进入沉默，而不是死亡（如果套接字干净地销毁，SSE 重连机制会第一个处理）。一种实现方式是，在 fx_server.id.php 文件的无限循环最上面加上下面这行代码：

```
    if($t % 10 == 0){sleep(45);break;}
```

换句话说，就是每 10 秒休眠很长一段时间，然后退出循环。（客户端会在 `sleep()` 结束之前断开连接，这样 PHP 会被关闭，因此 `break` 在这里并不需要。）如果用那种方法，当重连时会引发一个问题，因为 `$t` 会除以 10，所以会周而复始地立即失败。变通的方案是在进入无限循环之前加上下面这行代码，这只是提前了除以 10 的时机：

```
while($t % 10 == 0 || $t % 10 == 9)$t += 0.25;
```



要看已经写好的代码，参见本书源码的 fx_server.die_slowly.php 文件，它与 fx_client.die_slowly.html 配套使用（与 fx_client.id.html 相比，唯一变动的地方是要连接的 URL）。

当测试这段代码时，会看到数据传输几秒后停止了。20 秒后（长连接计时器的时长），它又重新连接，并且数据刚好接上断开连接时的位置。（参见 5.3.3 节，找出非正常地销毁套接字的方法，然后看看代码如何处理这种情况。）

## 5.9 不要全局化，考虑本地化

到目前为止，代码中已经使用了大量的全局变量，附录 B 会介绍为什么这样不好以及如何改进，但重点是使用全局变量不利于代码复用：不能在一个页面中使用多个 SSE 连接。下面的代码源自 fx_client.id.html，粗体部分是增加的代码：

```
var url = "fx_server.id.php?";

function SSE(url,options){
if(!options)options={};
var defaultOptions={
    keepaliveSecs: 20
    };
for(var key in defaultOptions)
    if(!options.hasOwnProperty(key))
        options[key]=defaultOptions[key];

var es = null;
var fullHistory = {};
var keepaliveTimer = null;
var lastId = null;

function gotActivity(){
if(keepaliveTimer != null)
    clearTimeout(keepaliveTimer);
 keepaliveTimer = setTimeout(
    connect, options.keepaliveSecs * 1000);
}
.
. (all other functions untouched)
.
```

```
    connect();
    }

    setTimeout(function(){new SSE(url);}, 100);
```

有两个主要的变化。

- 封装所有的变量和函数，这样 SSE 就是唯一的全局变量，这就可以创建多个实例。
- 引入 `options` 参数，这样一切都是可配置的，`keepaliveSecs` 被移到了这个位置。

我说过可以创建多个实例，但只创建 2 个实例而不考虑数据及其展示数据是不合理的。现在，代码经过了硬编码，以使用 fx_client.closure.html 中的静态 HTML。所以两个实例会在控制 HTML 上有冲突。该怎么做？如果想以一个 HMTL 表格来展示来自两个不同源的合并数据（比如，美元 / 日元来自一个数据源，欧元 / 美元来自另一个数据源），应该在 SSE 构造函数中取出 `fullHistory` 后，返回给一个全局变量，并伴随有 `updateHistoryTable()` 和 `makeHistoryTbody()`。另一方面，如果要在浏览器中展示两套数据，需要把每块 HTML 分别包裹在一个 `div` 中，并且把 `div` 的 ID 作为参数传给 SSE 对象（参见附录 B“两杯茶和两种茶”，这是后面这个方案的例子）。

## 5.10　阻止缓存

浏览器在是否应该缓存数据流方面会有些不确定。但是，稍微明显一点并无大碍。所以，在脚本的最上面（靠近设置 `Content-Type` 请求头的地方）添加下面两行代码：

```
header("Cache-Control: no-cache, must-revalidate");
header("Expires: Sun, 31 Dec 2000 05:00:00 GMT");
```

第一行是 HTTP/1.1 规范，本来只需要这一行就够了，因为这个规范是在 1999 年定义的。但是，现在仍然还有一些旧的代理服务器，第二行就是起这个作用的，设置成任何一个过去的日期即可。也可以再添加一行 `header('Pragma: no-cache');`，但这对新老浏览器、服务器以及代理服务器都是完全多余的。

## 5.11　阻止死亡

这段代码是 PHP 专有的，并且在 Windows 上比在 Linux 上更重要。如果脚本挂掉了 30 秒，这段代码正好可以修复，附录 C 的 C.6 节介绍了为什么要这么做。只需把这段代码放到脚本的最上面（刚刚在 `date_default_timezone_set('UTC');` 这一行后面就好）：

```
set_time_limit(0);
```

## 5.12 精简的简单办法

摆脱臃肿的简单办法，只需下面这粒灵丹妙药就可美梦成真：

```
AddOutputFilterByType DEFLATE text/html text/plain text/xml text/event-stream
```

如果插入到正确的位置（比如，Apache 服务的配置文件），它会对返回的数据进行 gzip 压缩。但现在配置中查找相似的语句，可能只需要添加 text/event-stream 到已有的配置中。比如，在 Ubuntu 中，有一个 deflate.conf 文件（在 /etc/apache2/mods-enabled/ 目录下），只需在含 `text/plain` 的那一行的后面加上 `text/event-stream`。

另一种配置 Apache 的方式是，除了一些图片格式外，将其他全部都 `DEFLATE`。如下所示（如果已经是这样了，那就没有什么需要为 SSE 而添加的）：

```
<Location />
    SetOutputFilter DEFLATE
    SetEnvIfNoCase Request_URI \.(?:gif|jpe?g|png)$ no-gzip dont-vary
    Header append Vary User-Agent env=!dont-vary
</Location>
```

更多关于 Apache 压缩方面的配置参见 http://httpd.apache.org/docs/2.4/mod/mod_deflate.html。添加 `Vary` 请求头是为避免一些代理浏览器的 bug。

如果用 IIS 作为网络服务器软件，关于如何为动态内容配置压缩可以参见这篇文章：http://technet.microsoft.com/en-us/library/cc753681.aspx

如果使用 nginx，参见 http://nginx.org/en/docs/http/ngx_http_gzip_module.html。注意，可能需要把 `gzip_min_length` 设成 0，或者比较小的值，以确保它也适用于流式传输的数据。

## 5.13 本章回顾

本章已经开始尝试改进应用的质量，方法包括添加错误报告，发送长连接消息，避免缓存问题以及出现问题时重连。在重连的方案上，同时使用了 SSE 的内置 `retry` 机制和我们自己的方案，它们都依赖于应用向我们发送断开连接前看到的最新数据的 ID。还介绍了定期关闭和支持多路连接。

接下来两章关注应用的使用范围，而不是质量，在保证本章介绍的产品级品质特性的前提下，允许不支持 SSE 的浏览器也能接收到一样的数据。

第 6 章

# 向后兼容：其他数据推送策略

本章要介绍的是一种叫长轮询的向后兼容解决方案，它（稍作调整）几乎可以适用于所有浏览器。如果数据推送的频率相对较低，那几乎察觉不到它的低效性，可以应用在任何地方，但通常只会在没有原生支持 SSE 的浏览器上使用这种方案。

本章和下一章会从最简单的示例代码开始介绍。然后，会对第 5 章末尾介绍的外汇对示例做一些修改以支持长轮询。在本章末尾，我们会使这个产品级品质的真实数据推送应用能够覆盖 99% 的浏览器（虽然有不同程度的低效）。

## 6.1　浏览器战争

从上世纪 90 年代中期开始，浏览器之间的差异（也就是“浏览器战争”）就一直困扰着我们。而当微软加入这场战争之后，情况变得尤其麻烦。我们从此进入了一个各大浏览器开发商通过单方面地添加特性的方式，使 Web 变得更好的时期。这是浏览器用户（就像你我这样的）和开发者们（还是你我这样的）都不愿意看到的局面。标准曾被讨论过而后又被忽视，只是最近几年，所有的浏览器开发商才开始认真对待标准。浏览器开发商最终意识到他们应该在用户体验和运行速度上做出特色，而不是做一些独有的特性。

但我们还是要处理他们造成的混乱。即便使用最新的 HTML5 技术，这种混乱依然存在，并且至少会持续 3 到 4 年。当 SSE 到来时，我会首先羞辱谷歌，然后是微软。Android 内置浏览器直到 Android 4.4 才开始支持 SSE（一些更早版本的 Android 设备上用 Chrome 可以支持 SSE）。XHR 向后兼容方案（下一章中介绍）适用于 Android 3 以后的版本，但在 Android 2.x 上不行（写本书时，仍然有大量 Android 2.x 的用户，参见 http://bit.ly/wiki-android-versions）。

庆幸的是，微软已经使 IE 的每个版本更多地迎合标准，所以 IE9 之后的版本勉强被大部分网络开发者们所接受。但是，到 IE11 都还没有支持 SSE，下一章的 XHR 向后兼容方案不适用于 IE9 及更早版本。还有另一种叫做 iframe 的向后兼容方案也会在第 7 章介绍，但它只适用于 IE8 以及更高版本。

本章介绍的长轮询方案效率上不及原生的 SSE（也不及第 7 章介绍的向后兼容方案），但对 Android 2.x 和 IE6/IE7 来说，这是唯一的选择。事实上，在大部分应用中，这种低效并不明显。但如果是每秒发送多个更新，可能其他资源（客户端的 CPU、服务端的 CPU、网络带宽）的占用会比较明显。



这并不是说不能在亚秒级的频率下使用长轮询，我试过用这种方案每秒发送 10 次更新，表现并没有太差。在每秒发送 100 次，并且发送一个 ID 指定最后收到的数据（参见 5.5 节）的情况下，这种方案还能跟上——可以获得所有的数据，但是一来一堆，不能很清晰地每秒获得 100 条数据。

## 6.2　什么是轮询

在介绍什么是长轮询之前，先介绍一下什么是常规轮询（如图 6-1 所示）。常规轮询好比你去敲好朋友家的门并且问她："准备好出去玩了吗？"，她会立即回答"好了"或者"没有"，如果她说"好了"，那你们就可以高兴地出去玩了；如果她说"没有"，当着你的面把门关上了，30 秒后你又来敲门并且再问一遍刚才的问题。最后，要么她准备好了，要么她并不是你真正的好朋友，要么你早上吃大蒜了。

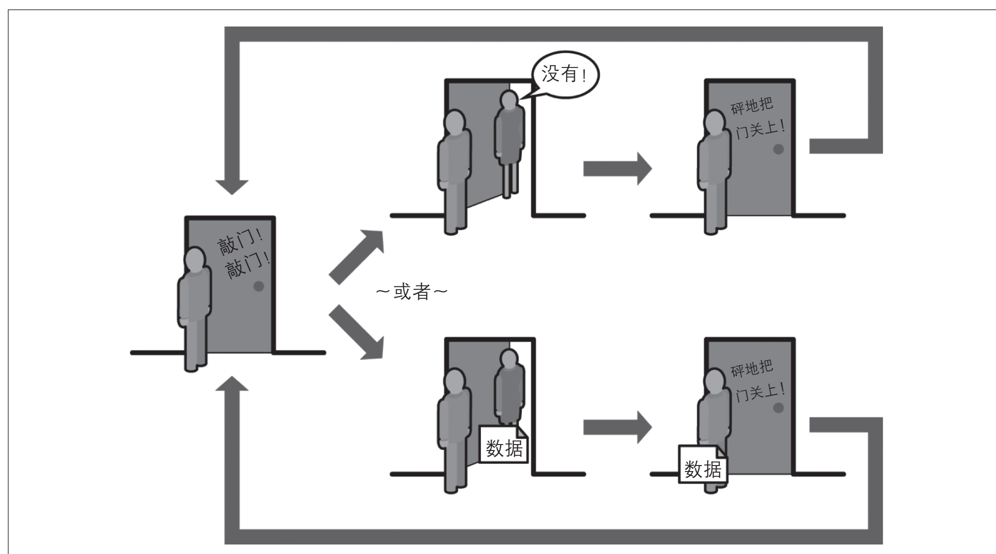



图 6-1：轮询你的好友

关于常规轮询，你需要了解的是，常规轮询就像你朋友一旦答应了，就会坐在那儿，像僵尸一样盯着墙，等着你过来再次敲门。

在我们这个外汇应用中，常规轮询意味着以一个固定的频率，比如每 10 秒一次，发送一个 Ajax HTTP 请求来询问数据。做常规轮询需要确定的是，是要做*采样*，还是要接收*一切*。如果是采样，服务端就发送所有外汇对的最新价格，每个外汇对对应一个价格。客户端会获取一个价格快照，而不是每次轮询请求的价格。采样的替代方案是，每次轮询时，将已接收到的最新数据的时间戳发送给服务端，然后请求那个时间之后的全部数据。如果没有新数据，服务端可能就返回一个空数组；如果有大量数据，服务端就会返回一个很大的数组。与采样相比，这种方案需要在客户端维护一个完整的历史记录数据。

# 6.3　怎样做长轮询

长轮询和常规轮询有什么不同呢？回到找好朋友玩的例子。我们去敲门然后问她："准备好出去玩了吗？"她回答："没有，但让门这样开着吧，我一准备好了就过来告诉你。"参见图 6-2，注意如何只敲一次门，门如何保持打开，以及*每次*访问如何获取数据。

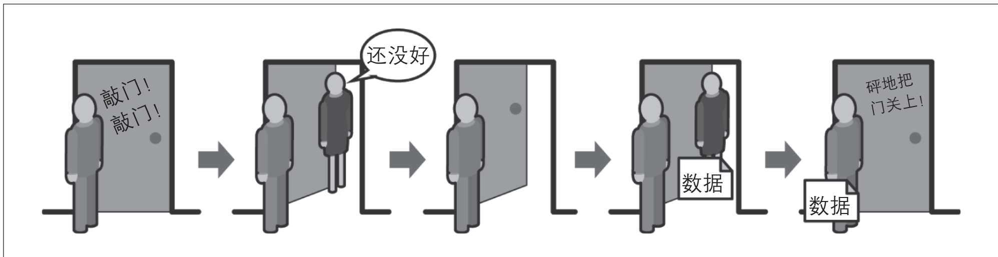



图 6-2：长轮询你的好友

对应到我们的应用中就是这样的：发送 Ajax HTTP 请求，但并不是询问最新的价格数据，而是要求下一次有新数据时被告知。如果没有数据更新，就一直这样打开一个套接字，然后一有新数据就立即发送，并关闭这个套接字。然后*立即*为下一个数据更新开始一个新的长轮询连接。

长轮询和 SSE 最关键的区别在于，需要为每一次数据更新新建一个 HTTP 连接。这也有和 SSE/WebSocket 一样的弊端，会几乎一直占用一个专门的套接字。在延迟方面，长轮询几乎表现得和 SSE 一样，一有新数据就能立即检测到，但也只是几乎一样好，因为每次要花费几毫秒新建一个 HTTP 连接。把数据更新的频率提高到每秒 10 次以上之后，所谓的"几毫秒"就会成为主要的延迟耗时（在移动网络等慢延迟网络上，所谓的几毫秒可能实际上是成百上千毫秒，所以它从一开始就慢了）。

### 长轮询是否总比常规轮询好

哪个更好取决于你怎么定义**更好**。从延迟的角度看，长轮询总是更好。当服务器有新数据时，长轮询能立即获取，而常规轮询则需要等到下一次轮询（SSE 和下一章要介绍的其他向后兼容方案也同样具备这种延迟方面的优势）。

但如果从带宽占用的角度来看呢？答案并不明确。如果外汇交易价格每秒更新两次，长轮询需要每分钟创建 120 个 HTTP 请求，如果用每 10 秒一次的常规轮询，每分钟只需要 6 个 HTTP 请求。所以常规轮询更好。但是，相反，如果外汇交易价格每分钟更新两次，长轮询只需要每分钟创建两个 HTTP 请求，而 10 秒一次的常规轮询仍然需要创建 6 个 HTTP 请求，并且延迟更糟！

关于数据的知识以及数据更新的确切时间，也能应用到长轮询或 SSE：当不需要任何数据的时候断开连接。这在延迟方面是一样的（假设能及时重连），但节省了套接字的占用（以及相关资源开销，比如 Apache 进程）。这是 5.4 节介绍过的技术（如果你使用这种方案，密切关注一下为什么重连会随机抖动）。

## 6.4 给我看些代码

说了很多，来点代码平衡一下，如何？首先来看后端代码：

```
<?php
usleep(2500000); //2.5s
header("Content-Type: text/plain");
echo date("Y-m-d H:i:s") . "\n";
```

这段代码相当简短。把这段代码保存为 minimal_longpoll.php 文件并放到网络服务器上，当调用它时，会有 2.5 秒的等待，然后展示当前时间戳。需要指出的是，是在休眠之后而不是之前发送请求头。休眠是为模拟等待下一次数据更新，并且在那之前并不知道会发送什么样的数据。比如，基于一些外部事件，可能最终需要发送一个错误码，这种情况下，代码就要改成如下样式：

```
<?php
usleep(2500000); //2.5s
$cat = (rand(1, 2) == 1) ? "dead" : "alive";
if ($cat == "dead") {
    header("HTTP/1.0 404 Not Found");
    echo "Something bad happened. Sorry.";
    } else {
    header("Content-Type: text/plain");
    echo date("Y-m-d H:i:s") . "\n";
    }
```

现在来看看前端代码。把 minimal_longpoll_test.html 文件放在与 minimal_longpoll.php 相同的目录下，并在浏览器中打开，会看到“Preparing!”在屏幕上闪烁一会儿，然后 JavaScript 开始运行并将它替换为“Started!”。再过一会儿被替换为 .[1]，这表明已经成功建立一个连接（`readyState==1`）。

两秒半之后会显示 .[1].[2].[3].[4]，后面跟着一个时间戳，然后在下一行，有另一个 .[1]（意味着另一个长轮询连接已经创建）。你所看到的显示效果会因浏览器不同而不同，这完全取决于 Ajax 的实现方式，只有 [1]（Ajax 请求开始）和 [4]（Ajax 请求完成）是重要的。想要了解这里的 1、2、3、4 分别是什么意思，可以参看本节附注内容"Ajax readyState"。

```
<!DOCTYPE html>
<html>
    <head>
        <noscript>
            <meta http-equiv="refresh"
                       content="0;URL=longpoll.nojs.php">
        </noscript>
        <meta charset="utf-8"/>
        <title>Minimal long-poll test</title>
    </head>
    <body>
        <p id="x">Preparing!</p>
        <script>
        function onreadystatechange() {
            s += ".[" + this.readyState + "]";
            document.getElementById( 'x' ).innerHTML = s;
            if (this.readyState != 4)return;
            s += this.responseText + "<br/>\n";
            document.getElementById( 'x' ).innerHTML = s;
            setTimeout(start, 50);
            }

        function start() {
            var xhr;
            if (window.XMLHttpRequest) {
                xhr = new XMLHttpRequest();
            }
            else {
                xhr = new ActiveXObject("Msxml2.XMLHTTP");
            }
            xhr.onreadystatechange = onreadystatechange;
            xhr.open( 'GET' , 'minimal_longpoll.php?t=' +
                (new Date().getTime()));
            xhr.send(null);
            }

        var s = "";
        setTimeout(start, 100);
        document.getElementById( 'x' ).innerHTML = "Started!";
        </script>

</body>
</html>
```

从 `start()` 函数开始研究这段源代码，这个函数里初始化了一个长轮询请求。首先创建一个 `XMLHttpRequest` 对象，如果是 IE 浏览器，就创建一个 Msxml2.XMLHTTP ActiveX 对象。这两个对象的函数和行为都是一样的，所以其他的代码都一样。下一行的 `xhr.`

`onreadystatechange = onreadystatechange;` 指定要调用的回调函数名。顺便插一句，我们可能用了 jQuery，从而不用关心 Ajax 复杂的创建过程。但这里还没有那么复杂，只是多了两三行代码而已。

`xhr.open` 用以指明从哪里获取数据，`xhr.send()` 则启动这个请求。（在一些浏览器上，需要显式地给 `send()` 传入一个参数 `null`）。

本章的开头曾提到，要让长轮询能在所有浏览器上运行需要费些周折。第一个要解决的问题就是一些浏览器（比如 Andorid 默认浏览器）会缓存 Ajax 请求。要避免这种情况，需要在 URL 上附加一点东西。一种简单的方式就是使用当前时间戳，也就是从 1970 年到现在的毫秒数。

对于 IE6/IE7，还要注意另一点：需要为每一个请求创建一个新的 XHR 对象。如果只创建一次 XHR 对象，然后每次发起一个长轮询请求时再调用一次 `send()`，这几乎在所有浏览器上都可运行，除了 IE7 以及更早版本。但是每次创建一个新对象，可以在任何浏览器上运行。在所有浏览器上都这么做，就不会再有什么麻烦。

另一个费点周折的地方是第一次调用 `start()`。需要用一个 `setTimeout()` 来增加一个 100 毫秒的延时，而不是直接调用。至少在一些版本的 Safari 上需要这样做。不这样做，就会一直处在加载状态，需要足够的时间让页面剩下的部分解析并达到就绪状态（Android 不需要，如果只需要在 Android 上支持长轮询，可以移除这个 100 毫秒的延时）。

接下来介绍 `onreadystatechange`（"on-ready-state-change"）函数。这是请求进度的回调函数（参见下面的附注内容）。这里只需要关心 `readyState` 变成 4 的情况，因为这意味着已经接收到新数据了，也意味着服务端已经关闭了这个连接。

### Ajax `readyState`

`XMLHttpRequest` 对象可以处在几个不同的状态（IE 浏览器的 `Msxml2.XMLHTTP` ActiveX 对象也一样）。通常你不需要关心，而且如果用 jQuery 创建 Ajax 连接，甚至都看不到这些状态码。状态码是从 0 到 4 的数字，它们的含义如下：

0

请求尚未开始。

1

已经与服务器连接上了。

2

请求（以及任何 POST 数据）已经发送到服务端。

3

数据获取中。

4

已获取所有数据并且已关闭连接。

在长轮询（以及短轮询和普通的 Ajax 的使用）中，我们忽略了除 readyState 变成 4 之外的情况。确切地说，onreadystatechange 是在 readyState 变成 4 的时候调用的，下一章会介绍一种需要考虑 readyState 3 的技术，onreadystatechange 会被调用多次。不同的浏览器处理的方式不同，有些浏览器会提供数据当前加载的进度信息。不同浏览器处理 readyState 值为 0、1 和 2 的情况不同，所以不能总是依赖这些值。

所以，每次调用这个函数都输出一个 .，但如果 readyState 还不是 4，就不做别的了。只要 readyState 变成了 4，就输出服务器发送过来的消息（在 responseText 中），然后通过调用 start() 开始下一个长轮询请求。

调用 start() 有一个 50 毫秒的延时，还是用 setTimeout() 来做，不然一些浏览器会搞混，并且最终报栈溢出错误。长轮询是为一些笨拙的浏览器做的兼容方案，所以大可不必为这多出的一点点延时而懊恼。（Android 还是不需要这个 50 毫秒的延迟。）

## 6.5 优化长轮询

前面提到长轮询大部分时候都表现很好，但是当事情变得一团糟时就开始低效了。如果每秒发送两次更新，那每分钟就要创建 120 个 HTTP 请求。在这种情况下，有两种方法可以用来稍稍降低负载。

第一种方法很简单：让客户端慢一点。事实上已经这样做了，在开始下一个长轮询之前有一个 50 毫秒的休眠。如果把这个休眠时间从 50 毫秒增加到 1000 毫秒，那每分钟最多要创建的长轮询请求是 60 个。考虑一些网络开销，每分钟最多达到 40 到 50 个。当数据的更新频率降低，其他的延迟问题就不那么重要了，获取下一次更新的时间是 16 秒以后而不是 15 秒以后，可以把休眠的时长想象为长轮询（零延迟，可能有大量的请求）的极限时间和常规轮询（可预见的延迟，可预见的请求频率）之间的连续。

另一种方案是在服务端，可以为长轮询的客户端缓冲数据，以不超过 1 秒 1 次的频率给客户端发送数据。这怎么工作？首先，记录下连接的时间（比如，18:30:00.000）。然后，假设在 18:30:00.150 时有数据要发送，这时先不发，设置一个 850 毫秒的延时，但在计时器触发之前（比如，在 18:30:00.900 时），又有要发给客户端的数据。继续等待，再等 100 毫秒，如果这 100 毫秒内没有新数据，那等 100 毫秒过后发送数据。这样，客户端会收到两份在一起的数据。

另一种情况，如果客户端在 18:30:00.000 时建立连接，但第一份数据在 18:30:01.100（请求开始后 1.1 秒后）才产生该怎么办呢？这种情况就立即发送数据并且关闭连接。换句话，人为的延时只会在 1 秒的时间内有多个数据时才会引入，这实际上意味着只会在有大量数据时放慢速度，这正是我们想要的。

做这个方案，建议把发送数据的最小时间间隔做成可方便配置的，这样就可以很容易地测试 500~2000 毫秒之间的值。

## 6.6 如果JavaScript被禁用怎么办

如果 JavaScript 被禁用了，那么本章所讲的都不会运行了。在这种情况下，运行我们最简单的示例时，屏幕上会一直是“Preparing!”，直到终老。这只不过是它们应得的。本书其他章节介绍的方案也都不会运行。

什么情况？你对这些用户心生怜悯？呸，骗人！不过，是有一种方法可以兼容这些 20 世纪的古董们。不需修改整个应用，只需修改 minimal_longpoll_example.html 文件，在 <head> 标签后添加下面的代码：

```
<noscript>
    <meta http-equiv="refresh" content="0;URL=longpoll.nojs.php">
</noscript>
```

因为包在一对 <noscript> 标签中，基本上不会运行，但是，那段代码的功能是在不支持 JavaScript 的浏览器上跳转到另一个页面，那个页面是 PHP 的，不是 HTML 的，那个 PHP 脚本需要生成完整的 HTML 页面，而不是像通过 Ajax 调用的脚本那样仅仅是发送该数据，它的代码相当简单，如下所示：

```
<!DOCTYPE html>
<html>
    <head>
        <script>window.location.href = "minimal_longpoll_test.html"</script>
        <meta http-equiv="refresh" content="3">
        <meta charset="utf-8"/>
        <title>Update test when JS disabled</title>
    <head>
    <body>
        <p><?= date("Y-m-d H:i:s"); ?></p>
        <p>(Enable JavaScript for better responsiveness.)</p>
    </body>
</html>
```

关键的一行是 <meta http-equiv="refresh" content="3">，意思是“3 秒后重载页面”。3 秒后会创建一个 HTTP 请求，这个 PHP 脚本会再运行一次并且创建一个新页面，显示一个新的时间戳。

还需要指出的是，文件顶部的 <script> 这一行是一个聪明的小技巧。如果用户只是临时禁用 JavaScript，那一旦启用 JavaScript，就会在下一次刷新页面时检测到，然后会回到具有完整实时更新服务的网站，那里有 21 世纪的人民在张开双臂欢迎。

# 6.7 将长轮询移植到我们的外汇交易应用

在第 5 章的最后，我们完成了一个相当强悍的示例应用。它能随机生成包含多个字段的外汇对（多路技术）数据，可以给所有接收到的数据维护一套历史记录，并且对那份历史记录做一些有意思的事，比如图表和表格。它会在出错时重连，并且记录下断开时的最后一条数据，也可以做定期的关闭和重连。

很幸运，把长轮询移植到我们的应用并不需要太多工作。事实上，简单是因为可以移植到一个分支处理的方法，而这是前面几章所做的所有细小的设计策略的结果。

## 6.7.1 连接

这个外汇交易应用现在有一个 SSE 对象，它包含了一个私有变量 es 和一个 startEventSource() 函数。第一个任务是为长轮询创建一个对等物[1]，下面是给 SSE 对象新增的私有变量：

```
var xhr = null;
var longPollTimer = null;
```

正如所示，这里也有一个保存计时器的变量（只会在 disconnect() 中用到）。下面是需要添加的函数：

```
function startLongPoll() {
    if (window.XMLHttpRequest)xhr = new XMLHttpRequest();
    else xhr = new ActiveXObject("Msxml2.XMLHTTP");
    xhr.onreadystatechange = longPollOnReadyStateChange;
    var u = url;
    u += "longpoll=1&t=" + (new Date().getTime());
    xhr.open("GET", u);
    if (last_id)xhr.setRequestHeader("Last-Event-ID", last_id)
    xhr.send(null);
}
function longPollOnReadyStateChange() {
    if (this.readyState != 4)return;
    longPollTimer = setTimeout(startLongPoll, 50);
    processNonSSE(this.responseText);
}
```

注 1：es 和 xhr 对象是互斥的，换句话说，如果浏览器使用 es，那 xhr 就是 null，如果浏览器用 xhr，那 es 就是 null。所以它们应该共用一个变量名，可能叫 server。我没有这样做，是为了强调它们各自代表一个不同类型的 JavaScript 对象。还有一个原因是，它们在关闭连接时调用的方法不一样，分别是 es.close() 和 xhr.abort()。

startLongPoll() 函数和它的 onreadystatechange 回调基本上和本章前面介绍过的函数一样，但有如下细微的区别。

- 使用全局的 url 变量，而不是将要连接的 URL 进行硬编码。
- 当设置好 last_id 之后，发送 Last-Event-ID 请求头（参见 5.5 节）。与 EventSource 不同的是，XMLHttpRequest 能发送 HTTP 请求头（IE 浏览器的 ActiveXObject 也可以），所以这里用到了这个功能。
- 数据处理过程会传送到函数 processNonSSE() 中，这个随后会介绍。
- longpoll=1 加到了 URL 中，这是让服务端知道在发送数据后关闭连接（记住，在长轮询方案中，不关闭连接浏览器收不到数据）。通过使用 longpoll=1，可以用一个后端支持多种前端兼容方案。
- 保存了计时器，所以用其他代码可以取消计时器。

还需要再添加一点东西，temporarilyDisconnect() 中有两个清理任务：

```
if(keepaliveTimer != null)clearTimeout(keepaliveTimer);
if(es)es.close();
```

可以添加 if(xhr)xhr.abort();，但在下一章还有更多要做，所以这里把所有相关语句都放到一个 disconnect() 函数中，并且在 temporarilyDisconnect() 中调用它，这两个函数就像下面这样：

```
function disconnect() {
    if (keepaliveTimer) {
        clearTimeout(keepaliveTimer);
        keepaliveTimer = null;
    }
    if (es) {
        es.close();
        es = null;
    }
    if (xhr) {
        xhr.abort();
        xhr = null;
    }
    if (longPollTimer) {
        clearTimeout(longPollTimer);
        longPollTimer = null;
    }
}
function temporarilyDisconnect(secs) {
    var millisecs = secs * 1000;
    millisecs -= Math.random() * 60000;
    if (millisecs < 0)return;
    disconnect();
    setTimeout(connect, millisecs);
}
```

## 6.7.2 长轮询和长连接

如果还记得 5.3.2 节的“客户端”，应该知道这里的长连接机制是当连接 20 秒内没有任何活动时调用 `connect()`。这在长轮询方案中会有问题，因为长轮询没有发送长连接消息的方式，它发完一条消息就断开了。好吧，当然，服务端会很高兴地发送长连接消息，但客户端收不到。



在那些 `ready State==3` 时会调用 `onreadystatechanged` 的浏览器中，可以接收到长连接消息。但是，如果可以那样做，那就可以用下一章会介绍的 XHR 技术，而不用麻烦现在的长轮询技术。

如果想了解这一点，可以参见本书源码的 longpoll_keepalive.php 和 longpoll_keepalive.html 文件。服务端每 2 秒发送一次长连接消息，然后 10 秒后发送真正的数据并退出。在每个浏览器上看看能收到什么，以及什么时候收到。在 Android 2.3 中（长轮询主要用于支持移动网络用户），会发现回调函数会在 `readyState==1` 时立即调用，然后在接下来的 10 秒什么也没有，最后状态码 2、3、4 都一起来了。

所以，如果长轮询在 20 秒内没有发送任何东西，发生了什么呢？不好的事。`startLongPoll` 又被调用，所以在服务端打开了两个套接字。如果服务端几个小时不发送任何东西，会打开上百个套接字。真的吗？几百个？差不多！记住，如果服务端在发送长连接消息，套接字会全部激活，所以不会被销毁。但不会是几百个，因为浏览器有并行连接数的限制，一般是 6 个。从某种意义上来说，这更糟：过不久之后，有 6 个长轮询连接打开，新请求会静静的放到一个栈中，并且对那个服务器的所有其他通信（比如，请求新图片）也都会被搁置。

通过添加下面加粗的两行代码，可以避免那种末日景象：

```
function startLongPoll() {
    if (xhr)xhr.abort();
    if (window.XMLHttpRequest)xhr = new XMLHttpRequest();
    else xhr = new ActiveXObject("Msxml2.XMLHTTP");
    xhr.onreadystatechange = longPollOnReadyStateChange;
    var u = url;
    u += "longpoll=1&t=" + (new Date().getTime());
    xhr.open("GET", u);
    if (lastId)xhr.setRequestHeader("Last-Event-ID", lastId)
    xhr.send(null);
}

function longPollOnReadyStateChange() {
    if (this.readyState != 4)return;
    xhr = null;
    longPollTimer = setTimeout(startLongPoll, 50);
    processNonSSE(this.responseText);
}
```

当带着数据成功地调用 onreadystatechange 函数时，xhr 被设置成 null，这与 startLongPoll() 中的第一行搭配。在那行代码中，如果 xhr 没有被设置成 null，就会调用 abort()。在一般操作中，当进入 startLongPoll() 时，xhr 会一直是 null。只有在因为长连接计时器触发调用时，xhr 不是 null，代表着上一个连接。换句话说，如果长轮询请求在 20 秒内没有应答，就会取消它并新建一个请求。

高兴吧？我不高兴。长轮询变得不那么像长轮询了。每 20 秒新建一个连接，这个代价不菲。好吧，长轮询时永远不用长连接如何？为了理解这好不好，想一下用长连接的初衷。

- 为了阻止中转服务器或者路由器关闭套接字。
- 为了在初始请求失败时持续重试。
- 为了在后端出错但套接字依然保持打开的情况下进行检测。（这也包含了浏览器出现间歇性的 bug 的情况。）

第一点要讨论的是是否要每 20 秒主动关闭套接字。但第二点和第三点理由充分，在一个产品的系统里不能没有这些。第三点有些麻烦：基本没有办法区分服务器的沉默是因为没有数据要发送，还是因为进入了一个死循环永远不会应答。建议在使用长轮询时，用一个更大的长连接计时器时间，因为谁也不希望发生那种崩溃，对吧？可以简单地加上下面这行代码：

```
function startLongPoll(){
keepaliveSecs = 300;
if(xhr)xhr.abort();
...
```

关于第二点（当初始请求失败时重试），参见下一节。

## 6.7.3 长轮询和连接错误

前面的长轮询代码写得相当乐观，理所当然地认为 URL 都是对的，并且服务器总是能接收请求。如果服务器因为任何原因掉线了呢？或者 URL 错了？任何一种情况下，longPollOnReadyStateChange 都会很快被调用，并且 readyState==4。可以通过 xhr 对象的 status 字段来验证，常见的 status 码的值如表 6-1 所示。

表6-1：常用的XMLHttpRequest状态码

| 状态码 | 含　义 |
| --- | --- |
| 0 | 连接问题，比如错误的域名 |
| 200 | 成功 |
| 304 | 来自缓存 |
| 401 | 身份验证失败 |
| 404 | 服务器存在，但找不到文件 |
| 500 | 服务器错误 |

希望永远不会看到 304，因为那将彻底摧毁流式实时数据！401 会被浏览器拦截，要求用户提供整数，然后再发送一次请求。仅当用户点击取消时，代码才会收到 401 状态。因此，除了“200”，其他状态码都会被当成错误处理。所有的错误中，都假设没有发送非法数据，然后在休眠 30 秒[2]后再发一次长轮询。只有当状态码是 200 时才使用数据，并立即再发一个长轮询。以下是修改之后的 `onreadystatechange` 回调函数：

```
function longPollOnReadyStateChange() {
    if (this.readyState != 4)return;
    xhr = null;
    if (this.status == 200) {
        longPollTimer = setTimeout(startLongPoll, 50);
        processNonSSE(this.responseText);
    }
    else {
        console.log("Connection failure, status:" + this.status);
        disconnect();
        longPollTimer = setTimeout(startLongPoll, 30000);
    }
}
```

`disconnect()` 函数停止了两个计时器（`longpollTimer` 和长连接计时器）以确保在 30 秒内没有其他地方调用 `startLongPoll()`。



如果想要变聪明，有一些状态码包含了有效信息。比如，301 意思是需要使用一个新的 URL。305 意思是应该使用一个代理，如果连接一个第三方系统，可能需要处理这些状况，希望它们能告知我们具体连接哪个代理。要注意 420 和 429，那意味着连接请求太频繁了。

## 6.7.4 服务器端

需要在之前版本的服务端脚本（fx_server.id.php）基础上做一些修改。首先是要指定长连接，在脚本的最上面，看看客户端是否在请求 `"longpoll"`：

```
$GLOBALS["is_longpoll"] = array_key_exists("longpoll",$_POST)
    || array_key_exists("longpoll",$_GET);
$GLOBALS["is_sse"] = !($GLOBALS["is_longpoll"]);
```

给 `$is_longpoll` 赋值为 `true` 或者 `false` 的这个语句很优雅紧凑。它只需检测输入数据（不论是 GET 数据还是 POST 数据）中 `longpoll` 是否存在，而不检测它的值。第二行代码的意思是如果不是长轮询，那必然是 SSE。其他的修改在主循环的最后面：

注 2：这里假设使用长轮询时，发送长连接消息的时间间大于 30 秒，参见 6.7.2 节，不然长连接计时器会首先触发，那会被这里的 30 秒超时干扰，这不太好。

```
..
if($GLOBALS["is_longpoll"])break;
}
```

简短有力，就像一只雄性蜜蜂，交付了包裹后就自杀。



这里显式地使用了 `$GLOBALS[]` 数组。这段代码和主循环在同一个作用域中（全局作用域），本来可以简单一点直接给 `$is_longpoll` 变量赋值，但显式地使用 `$GLOBALS[]` 意味着，当这段代码或者主循环封装到函数中时，代码仍然可以运行。同时也在说明这段代码，给半年后要维护这段代码的程序员一个显著的提醒："这些是全局变量"。

另一个修改将会在向后兼容方案中会用到。你可能还记得前面介绍过的下面这些辅助函数：

```
function sendData($data){
    echo "data:";
    echo json_encode($data)."\n";
    echo "\n";
    @flush();@ob_flush();
}
function sendIdAndData($data){
    $id = $data["rows"][0]["id"];
    echo "id:".json_encode($id)."\n";
    sendData($data);
}
```

使用向后兼容方案时，SSE 特有的那部分（`data:` 前缀，末尾额外的空行，以及单独的 `id:` 行）是不需要的，而事实上它们有些碍事。那为何不去掉呢？

```
function sendData($data) {
    if ($GLOBALS["is_sse"]) echo "data:";
    echo json_encode($data) . "\n";
    if ($GLOBALS["is_sse"]) echo "\n";
    @flush();@ob_flush();
}

function sendIdAndData($data) {
    if ($GLOBALS["is_sse"]) {
        $id = $data["rows"][0]["id"];
        echo "id:" . json_encode($id) . "\n";
    }
    sendData($data);
}
```

这意味着对长轮询来说 `sendIdAndData()` 和 `sendData()` 现在是一样的。这很好。可以在本书源码的 fx_server.longpoll.php 找到这部分代码（如果服务端代码发送 `retry:`，也需要做同样的事）。



如果想要做一个兼容，不要这样做，而应该在客户端截掉“data:”那一行的“data:”，并且要忽略其他行。

下面是最后一个修改。将下面这段代码：

```
header("Content-Type: text/event-stream");
```

替换成[3]：

```
if($GLOBALS["is_sse"])header("Content-Type: text/event-stream");
else header("Content-Type: text/plain");
```

### 6.7.5　处理数据

回到前端以及前面介绍过的 `processNonSSE()` 函数。这个函数在下一章也会用到，它会做两件使用 SSE 时会由浏览器做的工作：

```
function processNonSSE(msg) {
    var lines = msg.split(/\n/);
    for (var ix in lines) {
        var s = lines[ix];
        if (s.length == 0)continue;
        if (s[0] != "{") {
            s = s.substring(s.indexOf("{"));
            if (s.length == 0)continue;
        }
        processOneLine(s);
    }
}
```

为了把第一项工作看得更清楚，下面提供一个简化的版本：

```
function processNonSSE(msg) {
    var lines = msg.split(/\n/);
    for (var ix in lines) {
        processOneLine(lines[ix]);
    }
}
```

SSE 协议总是一次给回调函数一条消息，长轮询可能会一次给多条消息[4]。所以前面的代码

注 3：你是否认为 `Content-Type` 应该是 `application/json` 而不是 `text/plain`？我们此处写的代码是为那些处在崩溃边缘的浏览器设计的一种变通方案。现在不是语义化网络演说的时刻。更严格地说，这里发送的数据并不完全是 JSON 格式。将多条数据一起发送时，数据其实是两条以 LF 隔开的 JSON 字符串。每一条都只在 `processOneLine()` 函数中转化为 JSON 对象。

注 4：好吧，fx_server.longpoll.php 没有发送多条消息。但它可以，参见 6.5 节。在下一章，会发送多条消息，不论我们是否愿意。

把它们分割成独立的行，然后就可以分别处理它们，但这个简化的版本太不成熟而且危险。

我们的应用协议确实是每条消息一个 JSON 对象，也必须是一行（CR 和 LF 需要转义成 JSON 格式）。但还记得 SSE 协议吗？它用一个空行结束一条消息。所以接下来要做的事是找到空行（`if(s.length == 0)`）并将它们清除（continue）。

`if(s[0]!="{")` 这个代码块是做什么的呢？这是脏数据防御。`process_one_line()` 期待收到 JSON 格式字符串，完整的 JSON，并且只是 JSON。如果是其他什么东西，解析时会抛出一个异常。事实上，它期望参数是一个 JSON 对象字符串，这意味着字符串必须以 { 开始，以 } 结束。如果在花括号左侧有任何别的东西（`if(s[0] != "{")`），`s.substring(s.indexOf("{"))` 会把它截掉，如果截掉之后什么都不剩了，那就完全忽略（顺便说一下，这个特别的脏数据防御会在 7.3 节中加以介绍，我还没看到长轮询会触发它）。

## 6.7.6 接起来

最后一步很简单，把下面加粗的一行添加到 connect() 中，然后可以在不支持 SSE 的浏览器上测试一下：

```
function connect() {
    gotActivity();
    if (window.EventSource)start_eventsource();
    else startLongPoll();
}
```

如何在支持 SSE 的浏览器上测试长轮询（或者其他后面要介绍的兼容方案）呢？建议加一些临时代码，而不是注释掉 SSE 相关代码，如下所示：

```
function connect() {
    gotActivity();

    if (true)startLongPoll(); else // 临时代码

    if (window.EventSource)start_eventsource();
    else startLongPoll();
}
```

我喜欢在前后都加上空行和注释，让它更突出，以免遗忘（在 fx_client.longpoll.html 文件中有这段强制使用长轮询的代码，可以把它移除来检验一下）。

## 6.7.7 IE8及更早版本

到目前为止，我们创建的代码几乎可以在任何浏览器上运行，包括 Android 2.x 的内置浏览器。但是，这种代码在 IE8 上无法运行。IE8 唯一的问题是不支持 `Object.keys`（用于 4.4 节介绍过的 `makeHistoryTbody()` 函数中）。可以把下面这段代码插到 `<head>` 标签内来使 IE8 支持：

```
<script>
    Object.keys = Object.keys || function (o, k, r) {
        r = [];
        for (k in o)if (o.hasOwnProperty(k))r.push(k);
        return r;
    }
</script>
```

如果浏览器原生支持 `Object.keys`，就会使用 `Object.keys=Object.keys`，否则就会给 `Object.keys` 赋予一个简单的函数。这个函数会遍历传入对象的属性，并把它们添加到一个数据组中。调用 `hasOwnProperty` 是为避免遍历到 `Object` 原型的属性。想要更深入地了解，可以上网搜一下或者找一本 JavaScript 方面的书看一看。

## 6.7.8 IE7及其更早版本

IE6、IE7 和 IE8 都不支持 `Object.keys`，而 IE6 和 IE7 连 JSON 都还不支持。现代浏览器（包括 IE8 及其以后版本）都内置了 JSON 对象，并提供了 `parse()` 和 `stringify()` 方法。我们的代码只需用到 `JSON.parse()`，所以如果很在意带宽，可以放弃这套方案。不过，这只会影响 IE6 和 IE7 的用户。他们肯定很高兴能找到一个仍然支持他们浏览器的网站，而不会介意额外加载一个文件，这里会使用已经可用的 json2.js 文件。



这个文件可以在本书源码中找到，也可以从 https://github.com/douglascrockford/JSON-js 获取。

事实上我在使用最小化的版本，文件大小从 17 530 字节缩减到了 3377 字节。

现在，IE6 和 IE7 的用户占比很小，没有必要让绝大多数用户也来下载一个他们不需要的补丁文件（加载了也没有关系，因为这会在不支持 JSON 的浏览器中创建 JSON 对象，但这是在浪费带宽）。这里用到了 IE 特有的版本检测，它是 IE 才有的特性（IE10 之后移除了这个特性），但这很适用我们的目的：

```
<!--[if lte IE 7]>
<script src="json2.min.js"></script>
<![endif]-->
```

IE7 及其更早版本会处理那段 `<script>`，然后加载并运行 json2.min.js。IE8 和 IE9 会处理 `<!--[if lte IE 7]>...<![endif]-->` 命令，但不会做任何事情，其他浏览器直接把它当成一个注释完全忽略。

总地来说，对于所有现代浏览器，包括 IE9 及之后的版本，这里浪费了 198 字节来给 IE8 及其更早版本打补丁。IE6 和 IE7 需要额外加载 3377 字节。

## 6.8 蜿蜒曲折的轮询

本章介绍了可以用作 SSE 替代方案的一种更原始的机制。如果只需要数据快照（相对于完整历史记录来说），或者延迟不是很重要（比如，如果可以接受每 5 分钟收到一批“过期的”数据，这样客户端就不会一直占用一个打开的套接字），常规轮询有时可能比 SSE 更好。然后我们介绍了长轮询。它最大的优势是可以在任何支持 Ajax 的操作系统 / 浏览器上运行，而如今这种操作系统 / 浏览器几乎遍地都是。它的劣势在于要为每一条消息新建一个 HTTP 请求。好消息是，在一些浏览器上有更高效的选择，这是下一章要讨论的主题。

第 7 章

# 向后兼容：另辟蹊径

上一章介绍了长轮询，我们将它视为一种从服务端向不支持 SSE 的客户器推送数据的方案。与 SSE 相比，它的优势在于几乎可以在任何地方运行；劣势在于处理频率较高的数据更新时，比 SSE 稍微多出的那一点延迟和带宽会变得很显著。本章会介绍两种替代方案，它们在延迟和带宽方面的表现几乎和 SSE 一样好。

第一种方案像长轮询一样会用到 Ajax，但会用 `readyState == 3` 而不是 `readyState == 4`。概括来说，这意味着会在连接还没关闭时一块一块地接收服务器推送的每一条数据，而不是像长轮询那样要等到关闭连接才能接收到数据（如果你之间跳过了 6.4 节的附注内容“Ajax `readyState`”现在可能正是回顾 Ajax `readyState` 值含义的好时机）。

这是个很好的方案，效率只比 SSE 低一点点，所以有点讽刺意味的是，这种方案却并没有覆盖多少桌面浏览器 [1]。为什么？因为大部分支持它的浏览器已经原生支持 SSE 了！但是，这种技术适用于 Android 4.x（写作本书的时候，它覆盖了大约 2/3 的 Android 用户）。

第二种向后兼容方案是专门面向 IE8 及以上版本的。这个方案里没有任何特别的 IE 专有特性，所以奇怪的是，它不能在 Firefox、Chrome、Safari 和 Opera 上运行，或者需要一些特殊处理才能运行。但那些浏览器已经原生支持 SSE，谁还在意这个呢？关键在于这项技术支持 IE8/IE9/IE9，这使应用的浏览器覆盖率又提高了 28% [2]。

注 1：仅 Firefox 3.x 和 Safari 3.x 支持该方案。

注 2：这是写作本书时全球范围的统计，你也可以说第 6 章介绍的长轮询方案已经支持了 IE8+ 浏览器。确切点说，它给那 28% 的用户提供了几乎和 SSE 一样高效的解决方案。

## 7.1 共性

像在第 6 章一样，在将技术移植到外汇交易应用之前，先用一个极简示例来介绍这些技术。本章将要介绍的两种技术（XHR 和 iframe）会使用同一个后端脚本，参见本书源码的 abc_stream.php 文件，代码如下所示：

```
<?php
header("Content-Type: text/plain");

if (array_key_exists("HTTP_USER_AGENT", $_SERVER)
    && strpos($_SERVER["HTTP_USER_AGENT"], "Chrome/") !== false)
    echo str_repeat(" ", 1023) . "\n";
@ob_flush();@flush();

$ch = "A";
while (true) {
    echo json_encode($ch . $ch) . "\n";
    @ob_flush();@flush();
    if ($ch == "Z") break;
    ++$ch;
    sleep(1);
    }
?>
```

这里输出 MIME 类型为 `text/plain`。注意，不能像对 SSE 一样使用 `text/eventstream`，因为不支持 SSE 的浏览器不能识别这种类型，会询问用户是否要把它保存为一个文件！

接下来的一行输出一个刚好 1024 字节的空白。只有 Chrome 浏览器需要用到这行代码，所以这里用了 `user-agent` 来检测（`array_key_exists("HTTP_USER_AGENT",$_SERVER)`，判断是否指定了 `user-agent`，然后用 PHP 中管用的字符串片段匹配 `strpos($string,$substring)!== false` 来判断 `$string` 中是否包含 `$substring`）。

后面的代码就简单了：输出 26 个字符串，每次输出间隔 1 秒。26 秒后关闭连接（这样做是方便观察关闭连接时浏览器如何表现）。就像前面几章中创建的 SSE 代码那样，`@ob_flush();@flush();` 是为确保数据立即发送，而不是进入缓冲区。



Chrome 自第 6 版就支持 SSE，并且它是自动更新的浏览器，所以实际上没有任何需要用到这种向后兼容技术的 Chrome 浏览器。但如果要在 Chrome 上运行本章的代码，那段生成 1024 字节空白的代码是必要的。这也是我想要介绍的，因为这是一种很有用的故障排除技术：当某个浏览器上出现问题时，一堆空白经常能创造奇迹 [3]。

注 3：另外，在空白前面加一些前缀会大不一样——这是一种强烈的暗示：要进行浏览器优化了，尤其是在缓存时。关于这一主题，有一种使 XHR 技术能在 Android 2.x 上运行的方法，而不仅仅是 Android 4.x！把 `echo json_encode($ch.$ch)."\n";` 这行代码（刚好输出 3 字节）修改为 `echo json_encode($ch.$ch).str_repeat(" ",1021)."\n";`（刚好输出 1024 字节）。是的，$2^{10}$ 字节。我觉得它看起来像一段缓冲，但这真的是一项令人讨厌的技术，因为发送的每一条消息都要进行填充。如果要发送的消息刚好那么大，并且在意延迟，在意带宽，并且发送数据的频率很高（意味着在 Andorid 2.x 上使用长轮询会让用户不悦），那么这可能是你想要的方法。其他情况下，在 Android 2.x 上最好使用长轮询。

顺便说一下，本章示例中的这些缓冲技巧都不能在 Opera 12 上运行！（但 Opera 11.0 之后的版本都支持 SSE，所以可以无视这个问题。）

在前端，本章介绍的这两种技术的共同点在于，都不会在后端每次发送新消息时都接收到新消息。相反，会创建一个很长的字符串来保持自连接开始的所有消息。这个字符串会随着时间的消逝而变得越来越长，从而给我们带来两大挑战：

- 如何只提取新消息；
- 如何避免过多得占用内存。

对于第一个挑战，可以使用下面这个函数，s 是到目前为止所有接收到的数据，prevOffset 是目前读取字符串的索引（第一次调用时是 0），callback 是会处理消息的函数。这个函数会返回一个新的处理起始点，这也是下一次调用会赋给 prevOffset 的值。如果没有新数据，prevOffset 的输入值会被返回：

```
function getNewText(s, prevOffset, callback) {
    if (!s)return prevOffset;
    var lastLF = s.lastIndexOf("\n") + 1;
    if (lastLF == 0 || prevOffset == lastLF)return prevOffset;
    var lines = s.substring(prevOffset, lastLF - 1).split(/\n/);
    for (var ix in lines)callback(lines[ix]);
    return lastLF; // 下一次的起始点
    }
```

这也显示了另一件需要注意的事（SSE 有内置处理，而长轮询也永远不会有这个问题）：可能获得半条消息。回顾一下第 3 章，我们确定了一行一条 JSON 消息的协议，如果服务端发送消息 {"x":3,"y":4}\n，也总是收到消息 {"x":3,"y":4}\n。但这没有保证，可能接收到的是 {"x":3,"y。过一会儿之后，Ajax 回调再次被调用，这次接收到的是 ":4}\n，所以 s 等于 {"x":3,"y":4}\n。当然，一旦知道可能会发生这种情况，处理起来也很简单：只需在输入字符串中找最后一个 LF，并暂时忽略在那之后的内容。这就是 s.lastIndexOf("\n") 这段 JavaScript 代码做的事。（+1 是因为它返回了 \n 的索引，下一次需要从它的下一个字符开始。）

通过比较 prevOffset 和 lastLF，我们可以判断出是否有新数据（意味着至少有一整行新数据）。s.substring(prevOffset,lastLF-1) 只提取新数据。然后 .split(/\n/) 将它分割成一个数组。最后，就能为每一条数据调用一次回调函数。

如何应对内存溢出的挑战呢？这涉及简单地监控字符串的大小，而一旦它变得相当大了，就关闭连接并重连。可以在基于个案分析的基础上得出这个“相当大”的定义，但我倾向于使用 32 768，除非有更好的不用它的理由（什么是更好的理由？比如在发送大块的数据，32 KB 可能刚好是两到三条消息的大小）。这在 XHR 和 iframe 方案的实现中没有介绍，但会在本章后面移植到外汇交易应用时介绍。

## 7.2 XHR

如果你已经研究过长轮询的代码，那就没有多少要讲的新内容。这里准备了一个 XMLHttp-Request 对象（因为这段代码不会与 IE6 兼容，所以不用麻烦去检测 XMLHttpRequest 是否存在），连接到 abc_stream.php 文件，并且设置了 onreadystatechange() 函数。为了能在 Safari 上使用（就像在长轮询代码中做的那样），调用延时 50 毫秒的 send()，如果不这样做也可以，但鼠标指针就会一直在“转圈圈”。我们还给 xhr 对象添加了一个名为 prevOffset 的自定义变量。

现在详细地介绍一下 onreadystatechange 函数。它会做两件事。首先，每次被调用时都创建一条日志，附加到 <pre id="x">（用 <pre> 就能看到返回数据的原始格式）。顺便说一下，如果调用时不传入新数据内容，就会立即返回。onreadystatechange 函数的最后一行用了 getNewText()，这是本章前面创建的。它会将最新收到的数据填充到 <p id="latest"> 中，代码如下所示：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"/>
    <title>Simple XHR Streaming Test</title>
    <script>
        function getNewText(s, prevOffset, callback) {
            if (!s)return prevOffset;
            var lastLF = s.lastIndexOf("\n") + 1;
            if (lastLF == 0 || prevOffset == lastLF)return prevOffset;
            var lines = s.substring(prevOffset, lastLF - 1).split(/\n/);
            for (var ix in lines)callback(lines[ix]);
            return lastLF; // 下一次的起始点
        }
        function process(line) {
            document.getElementById("latest").innerHTML = line;
        }
    </script>
</head>
<body>
<p id="latest">Preparing...</p>
<hr/>
<pre id="x">Preparing...</pre>
<script>
    var s = "", s2prev = "";
    var xhr = new XMLHttpRequest();
    xhr.prevOffset = 0;
    xhr.open("GET", "abc_stream.php");
    xhr.onreadystatechange = function () {
        var s2 = this.readyState + ":" +
            this.status + ":" + this.responseText;
        if (s2 == s2prev)return;
        s2prev = s2;
        s += s2 + "<br/>\n";
```

```
            document.getElementById("x").innerHTML = s;
            this.prevOffset = getNewText(
                this.responseText, this.prevOffset, process);
        };

        setTimeout(function () {xhr.send(null)}, 50);
    </script>
    </body>
    </html>
```

所以，把 simple_xhr_test.html 和 abc_stream.php 放到同一个目录中，然后在支持的浏览器上打开，会看到下面的内容：

```
"CC"

1:0:

2:200:

3:200:"AA"

3:200:"AA"
"BB"

3:200:"AA"
"BB"
"CC"
```

26 秒之后，会在屏幕顶部看到 "ZZ"；在屏幕底部，会看到下面这两部分：

```
3:200:"AA"
"BB"
"CC"
"DD"
...
"YY"
"ZZ"

4:200:"AA"
"BB"
"CC"
"DD"
...
"YY"
"ZZ"
```

可以看到在 `readyState==3` 状态下接收到了 "AA" 到 "ZZ" 的所有内容。当后端服务关闭连接时，也会收到一个 `readyState==4` 的信号。

现在是时候指出，在大部分浏览器上，直接打开 abc_stream.php 文件会有令人惊讶的表现：不会看到 "AA"，然后 "AA" "BB"，而是 26 秒什么也不做，然后突然显示从 "AA" 到 "ZZ" 的所有内容。只有在用 `XMLHttpRequest` 时才会看到部分加载的数据。事实上，这也正是下一

节介绍的 iframe 技术不会在那些浏览器上运行的原因。

## 7.3 iframe

上一节介绍的 XHR 技术不能在 IE 上运行，原因是 IE 在 `xhr.readyState` 是 4 之前不会设置 `xhr.responseText` 的值！ XHR 技术的要点在于 `xhr.readyState` 永远不会成为 4，所以这是个致命的打击。不过并不是没有解决的办法。在 IE 中使用的技巧是把那些数据加载到一个动态创建的 `<iframe>`，然后去查看这个 iframe 的源码！第一次听到这个想法的时候，我兴奋得从椅子上跳起来去向我的猫解释。是的，人们表达兴奋的方式不一样，事实证明，猫也是。

这种向后兼容方案似乎可以在大部分浏览器上运行，而不仅仅是各种版本的 IE，但是不同浏览器在数据可用之前，要求从服务器接收到的数据量是不一样的。在 IE6/IE7/IE8 中，达到可用状态只需接收几字节数据，但是除非消息很短，否则不用担心这个。

但是为了在其他浏览器上运行本章的代码，需要做一些额外的特殊处理。Chrome 的要求似乎是 1024 字节，就像前面介绍的 XHR 技术一样。abc_stream.php 已经为 Chrome 发送了那么多的空白。Firefox 需要 2048 字节（所以刚开始我还不知道这套方案可以在 Firefox 上运行），而 XHR 技术不需要这么多。把下面加粗的代码添加到 abc_stream.php 中，就可以立即在 Firefox 中运行了：

```
header("Content-Type: text/plain");

if (array_key_exists("HTTP_USER_AGENT", $_SERVER)
    && strpos($_SERVER["HTTP_USER_AGENT"], "Chrome/") !== false)
    echo str_repeat(" ", 1023) . "\n";
if (array_key_exists("HTTP_USER_AGENT", $_SERVER)
    && strpos($_SERVER["HTTP_USER_AGENT"], "Firefox/") !== false)
    echo str_repeat(" ", 2047) . "\n";
@ob_flush();@flush();
...
```



记住，我没有把前面的代码用在外汇交易应用中，因为 Chrome 和 Firefox 绝不需要用到 iframe 或 XHR 兼容方案，它们总是会用原生 SSE。这些技巧只是为了不必使用 IE 来运行本节的代码。

你是否注意到，我随便提了一下在 IE6/IE7/IE8 中这些数据是可用的。等等，之前我不是说过这种技术只能在 IE8 上运行吗？问题在于 IE7 及更早版本不允许通过 JavaScript 访问嵌入的 iframe 的内容。下面这段代码可以测试是否可以访问 iframe 的内容：

```
if(window.postMessage){ /* OK */ }
```

windows.postMessage 在 IE8 及以上版本会返回 true，在 IE7 及更早版本会返回 false。



IE10 及更高版本的开发者工具有一个兼容模式，它允许浏览器模拟 IE9、IE8 或 IE7。当模拟 IE7 时，windows.postMessage 返回 true，意味着 iframe 方案似乎能在 IE7 上运行。我相信这是 IE10 的兼容模式的 bug 或局限性，没别的意思。

在一个特殊方面，iframe 技术要劣于 XHR 技术：我们必须拉取。但这和第 6 章介绍的拉取不同，因为不是从服务端拉取。而是从一个 iframe 中拉取变化，是完全本地的拉取，并且相对快捷轻巧，但它仍然有一些延迟。换句话说，服务器可以立即推送消息到客户端，但客户端需要花一点时间发现并处理新消息。这里的例子用了 setInterval(...,500)，那意味着每 500 毫秒查找一次新数据。所以，平均延迟是 250 毫秒。如果将 setInterval 的时间间隔减少到 100 毫秒，平均延迟就减少到 50 毫秒。缺点是客户端需要为额外的拉取占用更多的 CPU 资源。你必须权衡一下应用的延迟要求和客户端 CPU 占用要求。代码如下所示：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8"/>
        <title>Simple IFrame-Streaming Test</title>
        <script>
        function getNewText(s, prevOffset, callback) {
            if (!s)return prevOffset;
            var lastLF = s.lastIndexOf("\n") + 1;
            if (lastLF == 0 || prevOffset == lastLF)return prevOffset;
            var lines = s.substring(prevOffset, lastLF - 1).split(/\n/);
            for (var ix in lines)callback(lines[ix]);
            return lastLF; // 下一次的起始点
            }
    </script>
</head>
<body>
    <p id="latest">Preparing...</p>
    <hr/>
    <pre id="x">Preparing...</pre>
    <script>
    function connectIframe() {
    iframe = document.createElement("iframe");
    iframe.setAttribute("style", "display: none;");
    iframe.setAttribute("src", "abc_stream.php");
    document.body.appendChild(iframe);
    var prevOffset = 0;
    setInterval(function () {
        var s = iframe.contentWindow.document.body.innerHTML;
        prevOffset = getNewText(s, prevOffset, function (line) {
            document.getElementById("latest").innerHTML = line;
            });
        document.getElementById("x").innerHTML = s;
```

```
        }, 500);
    }

    if (window.postMessage) {
        document.getElementById("x").innerHTML = "OK";
        setTimeout(connectIframe, 100);
        }
    else {
        document.getElementById("x").innerHTML = "Sorry!";
        }
    </script>
    </body>
</html>
```

我们从这段代码的最下面开始查找 `window.postMessage`，如果存在，则调用 `connectIframe()`。这里必须要有 100 毫秒的延时，不然在创建 iframe 时会有一个 HTML 解析错误。在 `connectIframe` 中，前 4 行动态地创建了一个 `<iframe>`，通过设置 CSS 样式为 `display:none` 来使它不可见，`src` 设置为流式数据源。然后用 `setInterval` 来设置一个常规计时器，并且每 500 毫秒抓取 iframe 的内容。就像上一节介绍的 XHR 的例子，把目前为止收到的所有内容放到 `"x"` 元素中，只把最新的消息放到 `"latest"` 元素中。

把 simple_iframe_test.html 放到 abc_stream.php 所在的目录并且在 IE8 或更高版浏览器中打开，就可以看到 `"latest"` 元素的内容每秒更新一次。

## 7.4 将XHR/iframe移植到外汇交易应用

移植的步骤和第 6 章介绍的移植长轮询类似。有一些对后端的细微修改，并且添加了一些类似于本章前面介绍过的简单前端代码，以及把它们连接起来的特性检测代码。

### 7.4.1 后端的XHR

还记得下面这段取自第 6 章的代码吗？

```
$GLOBALS["is_longpoll"] = array_key_exists("longpoll", $_POST)
    || array_key_exists("longpoll", $_GET);
$GLOBALS["is_sse"] = !($GLOBALS["is_longpoll"]);
```

客户端（包括 XHR 和 iframe）会识别出自己在用 XHR，所以会对其进行修改，如下所示：

```
$GLOBALS["is_longpoll"] = array_key_exists("longpoll", $_POST)
    || array_key_exists("longpoll", $_GET);
$GLOBALS["is_xhr"] = array_key_exists("xhr", $_POST)
    || array_key_exists("xhr", $_GET);
$GLOBALS["is_sse"] = !($GLOBALS["is_longpoll"] || $GLOBALS["is_xhr"]);
```

服务器端没有什么要做的。推送数据的格式和使用长轮询是完全一样的。`xhr` 基本上只用于设定正确的 MIME 类型（(text/plain 而不是 text/event-stream，因为后者会使一些浏览器询问

用户是否要保存为文件)。(上面的代码可以在本书源码的 fx_server.xhr.php 文件中找到。)

### 7.4.2 前端的XHR

将下面这段代码添加到 fx_client.longpoll.html 文件中，就像上一章末尾的那个文件一样：

```
function getNewText(s, prevOffset) {
if (!s)return prevOffset;
var lastLF = s.lastIndexOf("\n") + 1;
if (lastLF == 0 || prevOffset == lastLF)return prevOffset;
var lines = s.substring(prevOffset, lastLF - 1).split(/\n/);
for (var ix in lines)processNonSSE(lines[ix]);
return lastLF; // 下一次的起点
}

function startXHR() {
if (xhr)xhr.abort();
xhr = new XMLHttpRequest();
xhr.prevOffset = 0;
xhr.onreadystatechange = function () {
    this.prevOffset = getNewText(
        this.responseText, this.prevOffset);
    };
var u = url;
u += "xhr=1&t=" + (new Date().getTime());
xhr.open("GET", u);
if (last_id)xhr.setRequestHeader("Last-Event-ID", last_id)
xhr.send(null);
}
```

`getNewText` 函数和之前看到的一样，但这里写死 `processNonSSE()` 作为回调函数，而不是把回调函数当做参数。这在 XHR 和 iframe 技术中都适用（应该还记得，它也用于长轮询）。`startXHR()` 函数和本章前面创建的简单示例类似，但它确实更加简单：没有报告 `xhr.readyState` 各种值的杂乱，用一行代码就可以处理一切。当 `readyState` 是 0、1 或者 2 时，`responseText` 是空的，所以 `getNewText` 不会做任何事情（并且返回 0）。注意，那是处理 `xhr.responseText` 是 `null` 的情况。`xhr` 是上一章定义的 SSE 对象的一个私有变量。

如果服务器关闭连接并且 `readyState` 是 4，那么会出现两种可能的情况。要么是上一次调用 `onreadystatechange` 之后没有新数据，要么有新数据（可能之前收到了半条消息，刚好在等剩下的部分和 LF）。不论哪种情况，`getNewText()` 都会做正确的事情。它是那种你可以带回家见家人的函数，不用担心它会让你尴尬。

### 7.4.3 前端的iframe

首先给 SSE 对象添加一个私有变量，就放在定义 `es` 和 `xhr` 的代码之后：

```
var iframe = null;
```



正如在前面一章里提到的那样，es、xhr 以及 iframe 是互斥的，这意味着它们可以都可以被称为 server，或者其他名称，并且共用相同的变量。为了使代码更清晰，本书选择使用 3 个独立的私有变量。

然后添加下面这个函数：

```
function startIframe() {
var u = url;
u += "xhr=1&t=" + (new Date().getTime());
iframe = document.createElement("iframe");
iframe.setAttribute("style", "display: none;");
iframe.setAttribute("src", u);
document.body.appendChild(iframe);
var prevOffset = 0;
setInterval(function () {
    if (!iframe.contentWindow.document.body)return;
    var s = iframe.contentWindow.document.body.innerHTML;
    prevOffset = getNewText(s, prevOffset);
    }, 500);
}
```

这基本上和本章前面介绍的代码一样，但是使用了全局的 URL，并删除了日志的代码。但它要稍微做一些优化，以达到产品品质。首先传送 lastId 变量（已经在 URL 里了，不是通过请求头）。接下来要介绍的一处修改是当第二次调用这个函数时，清理一下前一个调用（应该还记得长连接机制也需要这样做）：

```
function startIframe() {
if (iframe)iframe.parentNode.removeChild(iframe);
if (iframeTimer)clearInterval(iframeTimer);
var u = url;
if (last_id)u += "last_id="
    + encodeURIComponent(last_id) + "&";
u += "xhr=1&t=" + (new Date().getTime());
iframe = document.createElement("iframe");
iframe.setAttribute("style", "display: none;");
iframe.setAttribute("src", u);
document.body.appendChild(iframe);
var prevOffset = 0;
iframeTimer = setInterval(function () {
    var s = iframe.contentWindow.document.body.innerHTML;
    prevOffset = getNewText(s, prevOffset);
    }, 500);
}
```

这也需要添加一个私有变量：var iframeTimer = null;。

### 7.4.4　接通XHR

现在，connect() 函数如下所示：

```
function connect() {
    gotActivity();
    if (window.EventSource)startEventSource();
    else startLongPoll();
}
```

添加下面这行代码：

```
function connect() {
gotActivity();
if (window.EventSource)startEventSource();
else if (window.XMLHttpRequest &&
    typeof new XMLHttpRequest().responseType != "undefined")
    startXHR();
else startLongPoll();
}
```

浏览器检测有一点复杂。这里需要 XMLHttpRequest2 来支持运行。对该函数的第一部分修改会检测 XMLHttpRequest 是否定义。几乎每一个浏览器都会因为它而返回 true，因为这在 XHR 的第一个版本中已经定义了。标准设计者给 XHR 加了一堆新特性之后，没有设计一个增强的 XMLHttpRequest2 对象，而是仍然使用 XMLHttpRequest 这个名称。遗憾的是，他们也没有设计任何类型的版本号，而且还没有任何直接体现 XMLHttpRequest2 功能的对象可用。

这是在耍花招。所以，给我们的测试方式恰巧都是一样的：所有在 XMLHttpRequest 对象上定义了 responseType 属性的浏览器[4]，都可以在 readyState==3 时访问 responseText 中的数据。



出于测试的目的，要强制支持 SSE 的浏览器使用 XHR，将下面这段代码放到 connect() 的最上面：

```
if(true)startXHR();else
```

### 7.4.5　接通iframe

如果你觉得 XHR 的特性检测复杂，说明你不是什么都没看见。iframe 的特性检测分为两部分。第一部分在 HTML 文件的顶部。上一章介绍了 IE 专用的宏语言。这里用它在 IE9 以及更早版本的浏览器上设置一个值为 true 的 JavaScript 全局变量，其他浏览器则设为 false。（没有在 IE10 及之后的版本中使用 iframe，是因为它们支持 XHR，幸好 IE 宏语言在 IE10 及之后版本就不支持了！）在 HTML 文件中靠近 <head> 顶部的地方，添加下面粗体部分的代码（其他部分的代码在 fx_client.longpoll.html 文件中已经有了）：

注 4：好吧，是我在上面尝试使用过该属性的所有浏览器。记住，在现实世界中，这个向后兼容方案只能用于 Android 4.x，在 Android 4.x 上，这个特性检测可以正常工作。

```
<script>var isIE9OrEarlier = false;</script>
<!--[if lte IE 7]>
<script src="json2.min.js"></script>
<![endif]-->
<!--[if lte IE 9]>
<script>
isIE9OrEarlier = true;
</script>
<![endif]-->
<script>
Object.keys = Object.keys || function (o, k, r) {
    r = [ ];
    for (k in o)if (o.hasOwnProperty(k))r.push(k);
    return r;
    }
</script>
```

现在有了 isIE9OrEarlier 这个新的全局变量，在 connect() 中添加下面这几行代码：

```
function connect() {
gotActivity();
if (window.EventSource)start_eventsource();
else if (isIE9OrEarlier) {
    if (window.postMessage)startIframe();
    else startLongPoll();
    }
else if (window.XMLHttpRequest &&
    typeof new XMLHttpRequest().responseType != "undefined")
    startXHR();
else startLongPoll();
}
```

那段代码说得很直白：如果是 IE9 及更早版本，那就用 iframe（比如 IE8 和 IE9，因为只有它们定义了 window.postMessage 函数）或者长轮询（比如 IE5.5、IE6 和 IE7）。如果是 IE10 或者 IE11，则使用 XHR。



好人做到底，我应该告诉你，要强行在支持 SSE 的浏览器中测试 iframe，把下面这一行放到 connect() 的最上面：

```
if(true)startIframe();else
```

还有一个修改：在 disconnect() 函数中再加两段代码：

```
function disconnect() {
if (keepaliveTimer) {
    clearTimeout(keepaliveTimer);
    keepaliveTimer = null;
    }
if (es) {
    es.close();
    es = null;
```

```
    }
    if (xhr) {
        xhr.abort();
        xhr = null;
        }
    if (longPollTimer) {
        clearTimeout(longPollTimer);
        longPollTimer = null;
        }
    if (iframeTimer) {
        clearTimeout(iframeTimer);
        iframeTimer = null;
        }
    if (iframe) {
        iframe.parentNode.removeChild(iframe);
        iframe = null;
        }
    }
```

## 7.5 感谢内存

我忘了什么吗？肯定忘了什么。噢，我的记性[5]不太好了……对，就是它！内存管理！XHR 和 iframe 方案都会保存服务器发送的消息，它基本上就是一个不为人知的重大消息。这在 SSE 中没有问题，因为 EventSource 对象独立对待每一条消息；在长轮询中也不是问题，因为每条消息都是一个完整的链接。如果运行一个脚本足够长的时间，XHR 和 iframe 会有问题，数据缓冲会越来越大，直到拖垮客户端系统。

解决方案简单粗暴：当这个重大消息变得太大时，新建一个连接。这有一些缺点，坦率地说，SSE 相比 XHR 兼容方案最大的优势就是，它不存在这个问题。在检测这个缺点之前，来看一下代码。这涉及在 getNewText()（在 XHR 和 iframe 都有用到，但是原生 SSE 和长轮询没有用到）中添加了加粗的那部分代码：

```
function getNewText(s, prevOffset) {
var lastLF = s.lastIndexOf("\n") + 1;
if (lastLF == 0 || prevOffset == lastLF)return prevOffset;
var lines = s.substring(prevOffset, lastLF - 1).split(/\n/);
for (var ix in lines)processNonSSE(lines[ix]);

if (lastLF > 65536) {
    console.log("Received " + lastLF +
        " bytes (" + s.length + "). Will reconnect.");
    disconnect();
    setTimeout(connect, 1);
    }

return lastLF; // 下一次的起始点
}
```

注 5：中文的“记性”和“内存”在英文中是同一个词 memory。——译者注

换句话说，一旦缓冲区超过了 64 KB，就会断开连接，然后重连。延时 1 毫秒调用 `connect()` 是为避免递归调用时潜在的问题。

第一个要指出的缺点是，选择 64 KB 是随意的。示例应用达到这个大小需要大概 2.5 分钟。如果每条消息更大，或者消息发送更快，可能需要增加缓冲大小。如果所有用户都是桌面用户，可以把缓冲增大 10 倍甚至更多，即便是在移动设备上，64 KB 也不是太大。

第二个缺点是可能会丢失半截消息，记住之前介绍过的关于使用 `s.lastIndexOf("\n")` 来处理消息的讨论。这些半截消息会比较罕见（但愿如此），所以可以把这一句改成 `if(lastLF > 65536 && lastLF == s.length)`，告诉它等到一个干净的点再关闭连接。只需记住这一点：这意味着理论上它永远不会断开连接（引起内存问题）。

第三个问题是和在长轮询中一样的问题：可能在断开连接到下一次连接的这段时间里错过了一条消息。可是，如果发送接收到的 `lastId`（就像在示例中所做的那样），然后第二个和第三个缺点就抵消了：不会丢失任何东西，只是有一点低效。

## 7.6　把襁褓中的外汇交易应用放到床上

在前面的五章中，我们完成了外汇交易应用的开发：兼容 99% 以上的浏览器，为这些浏览器用户使用我们能够找到的最有效率的技术（如果忘了哪些浏览器用到哪些技术，可以参见下面附注内容中的表 7-1），为一个真实复杂的数据推送应用使用最高效的技术，这个应用处理了服务端和套接字离线、定期关闭等问题。

第 9 章介绍身份认证和其他安全相关的问题时，会再次使用这个外汇交易应用。在此之前，第 8 章将介绍那些示例应用中尚未用到的 SSE 的其他一些方面。

## 各自的归宿

表 7-1 总结了浏览器检测的工作原理。

表7-1：不同用户的浏览器该用哪个start()函数

| 函数 | 浏览器 |
|---|---|
| startEventSource() | • 基本上所有的 Firefox 和 Chrome [*]<br>• 桌面 Safari 5.0+<br>• iOS Safari 4.0+<br>• Android 4.4+（之前 Chrome 是默认浏览器）<br>• Android 版的 Chrome（所有版本）<br>• Android 版的 Firefox（所有版本）<br>• Opera 11.0 以上（含 11.0）<br>• 移动版 Opera 11.1 以上（含 11.1）<br>• 黑莓 7.0 以上（含 7.0） |
| startXHR() | • IE10+<br>• Firefox 3.6（以及更早版本）<br>• Safari 3.x<br>• Android 4.1 到 Android4.3（除非 Chrome 是默认浏览器）<br>• Android 3.x |
| startIframe() | • IE8<br>• IE9 |
| startLongpoll() | • IE6<br>• IE7<br>• Android 2.x<br>• 前面没有列出的其他任何支持 Ajax 的浏览器 |
| (none) | • 任何禁用 JavaScript 的浏览器 |

* 从严格意义来说，是从 Firefox 6 和 Chrome 6 开始的，但是 Firefox 自第 4 版开始有自动更新，而 Chrome 自 beta 版就有自动更新，所以有理由认为没人在用不支持 SSE 的版本。

# 第 8 章

# 关于SSE的其他标准

SSE 标准还有一些我之前简略谈到的其他特性，本章将对它们进行探讨。之前没有介绍这些特性有两个理由。首先，我们不需要用到它们！因为一行一条 JSON 字符串的设计决策，以及 JSON 对象的自描述性，事件和多行特性都用不到。第二个理由是，那会使向后兼容方案变得更慢更复杂。从实用性角度出发，而且无意创建可供下载的完美 SSE 代码，可以允许向后兼容方案使用最合适的协议。但是了解这些特性是个好事，本章会介绍每个特性，介绍如何使用它们，甚至提供了一些如何在向后兼容方案中实现的提示。

## 8.1 请求头

下面是一个简单的脚本（可以在本书源码的 log_headers.html 文件中找到）：

```
<html>
    <head>
        <title>Logging test</title>
    </head>
    <body>
        <script>
        var es = new EventSource("log_headers.php");
        </script>
    </body>
</html>
```

这显示了一个 SSE 脚本可以有多小。当然，这绝对没有在前端做任何事情。下面是与之对应的后端代码：

```
<?php
$SSE = (@$_SERVER["HTTP_ACCEPT"] == "text/event-stream");
if ($SSE)
    header("Content-Type: text/event-stream");
else
    header("Content-Type: text/plain");
file_put_contents("tmp.log", print_r($_SERVER, true));
?>
```

这段代码也短得让人不好意思了。它只是把超全局变量 `$_SERVER` 中的所有东西简单地写到一个 tmp.log 文件中。这里包含了浏览器发送到服务器的 HTTP 请求头，通常这是我们感兴趣的东西。tmp.log 只显示了最近的请求，每次都会被重写。可以在你的目标浏览器上试一下它。



如果通过网络服务器访问时没有创建 tmp.log 文件，可能是因为写权限的问题。如果服务器是 Unix 系统，在命令行运行 touch tmp.log 和 chmod 666 tmp.log 之后再试一下。

之所以首先介绍这个，是因为可以把 `file_put_contents("tmp.log",print_r($_SERVER,true));` 这一行放到任何脚本代码的上面，以诊断问题或只是帮助理解。

如果想看 COOKIES、POST 以及其他任何超全局变量的内容，尽管也把它们加到脚本里。但是，更好的方式是显示 `phpinfo()` 的输出。图 8-1 显示了这些内容的摘要，因为它是 PHP 特有的，这里就不介绍了。如果有兴趣，可以看一下 show_phpinfo.php 文件。

show_phpinfo.php 脚本抓取了 `phpinfo()` 输出（HTML 格式），并对其做了一点格式化，然后以一个 SSE 块输出。它包裹在一个 JSON 字符串中，以确保换行符不会引发问题（这段代码也适用于 XHR 和长轮询方案，也包含了一些本章出现的请求头，以使它更广泛地使用）。前端代码如下所示：

```
<html>
<head>
    <title>PHPInfo Test</title>
</head>
<body>
<div id="x">(loading...)</div>
<script>
    var es = new EventSource("show_phpinfo.php");
    es.addEventListener("message", function (e) {
        var s = JSON.parse(e.data);
        document.getElementById("x").innerHTML = s;
    }, false);
</script>
</body>
</html>
```

如果只看见“(loading…)”，可能是访问 show_phpinfo.php 时出现了“403 Forbidden”错误，下面的警告解释了原因。

| | | |
|---|---|---|
| apc.slam_defense | On | On |
| apc.stat | On | On |
| apc.stat_ctime | Off | Off |
| apc.ttl | 0 | 0 |
| apc.use_request_time | On | On |
| apc.user_entries_hint | 4096 | 4096 |
| apc.user_ttl | 0 | 0 |
| apc.write_lock | On | On |

**APD**

| Advanced PHP Debugger (APD) | Enabled |
|---|---|
| APD Version | 1.0.1 |

**bcmath**

| | |
|---|---|
| BCMath support | enabled |

| Directive | Local Value | Master Value |
|---|---|---|
| bcmath.scale | 0 | 0 |

**bz2**

| | |
|---|---|
| BZip2 Support | Enabled |
| Stream Wrapper support | compress.bz2:// |
| Stream Filter support | bzip2.decompress, bzip2.compress |
| BZip2 Version | 1.0.5, 10-Dec-2007 |

**calendar**

| | |
|---|---|
| Calendar support | enabled |

**Core**

| | |
|---|---|
| PHP Version | 5.3.2-1ubuntu4.21 |

| Directive | Local Value | Master Value |
|---|---|---|
| allow_call_time_pass_reference | Off | Off |
| allow_url_fopen | On | On |
| allow_url_include | Off | Off |
| always_populate_raw_post_data | Off | Off |
| arg_separator.input | & | & |
| arg_separator.output | & | & |
| asp_tags | Off | Off |
| auto_append_file | *no value* | *no value* |
| auto_globals_jit | On | On |
| auto_prepend_file | *no value* | *no value* |
| browscap | *no value* | *no value* |

图 8-1：show_phpinfo.html 的示例输出

> 不要把这段专用的脚本放到任何产品服务器上。phpinfo() 输出了系统非常详细的细节，有些可能会被黑客利用。
>
> 因为读者可能在读本章之前就把本书源码上传到了他们的网络服务器，里面

包含了一个专门阻止访问 show_phpinfo.php 的 .htaccess 文件，所使用的代码块如下所示：

```
<Files "show_phpinfo.php">
    deny from all
</Files>
```

在一个**你坚信外面的世界不会访问到**的系统中，大可从 g.htaccess 文件中删除那段代码。

注意，.htaccess 文件只会在 Apache 配置它们可用时才会生效，有时会因性能或中央控制的原因禁用。将 Apache 配置文件中 AllowOverride 改为 All，或者至少改为 AllowOverride AuthConfig Limit，就可以使 .htaccess 生效。更多信息参见 https://httpd.apache.org/docs/2.0/mod/core.html#allowoverride。

更多关于使用 .htaccess 文件控制 SSE 资源的访问，参见 9.2 节。

其他大部分语言同样支持访问请求头。下面是在一个单独的 Node.js 服务器上实现的代码：

```
var http = require("http");

http.createServer(function (request, response) {
    console.log(request.method + " " + request.url);
    console.log(request.headers);

    if (request.url != "/sse") {
        response.end("<html>" +
            "<head><title>Logging test</title></head>" +
            "<body><script>" +
            "var es = new EventSource('/sse');" +
            "</script></body></html>\n");
        return;
        }
    response.writeHead(200,
        { "Content-Type": "text/plain" });
    response.end();
    }).listen(1234);
```

运行 node log_headers.node.js 启动它，它会监听服务器上所有 IP 的 1234 端口。关键的一行是 `console.log(request.headers);`，它输出到控制台，但也可以很简单地修改成输出到一个日志文件，就像 PHP 例子所做的那样。

脚本剩下的部分是返回可以再次用 SSE 访问服务器的 HTML 文件框架。还有一行你可能有兴趣的代码：`console.log(request.method+" "+request.url);`，它显示了被请求的是哪个文件。

## 8.2 事件

正如在本书中所看到的那样，服务器给发送的数据加了 `data:` 前缀。然后客户端通过为

message 事件创建一个处理程序接收这些数据：

```
es.addEventListener("message", function(e){
    var d = JSON.parse(e.data);
    document.getElementById(d.symbol).innerHTML = d.bid;
},false);
```

结果表明，message 是默认的，可以用这种方式让服务器标记每一行，这样就可以在前端用一个不同的函数来处理。

标记方式是给一行数据加上 event: 前缀，然后在客户端通过指定这类事件的处理程序来处理数据。看一下例子就很清楚了。回到外汇交易应用，发送的数据可能如下所示：

```
event:AUD/GBP
data:{"timestamp":"2014-02-28 06:49:55.081","bid":"1.47219","ask":"1.47239"}

event:USD/JPY
data:{"timestamp":"2014-02-28 06:49:56.222","bid":"94.956","ask":"94.966"}

event:EUR/USD
data:{"timestamp":"2014-02-28 06:49:56.790","bid":"1.30931","ask":"1.30941"}

event:EUR/USD
data:{"timestamp":"2014-02-28 06:49:57.002","bid":"1.30983","ask":"1.30993"}

event:EUR/USD
data:{"timestamp":"2014-02-28 06:49:57.450","bid":"1.30972","ask":"1.30982"}

event:AUD/GBP
data:{"timestamp":"2014-02-28 06:49:57.987","bid":"1.47235","ask":"1.47255"}

event:AUD/GBP
data:{"timestamp":"2014-02-28 06:49:58.345","bid":"1.47129","ask":"1.47149"}
```

将这段代码与 3.5 节中的第一段代码对比一下。如果很在意带宽，像这样使用 event:，每条消息好像可以节省 6 字节。然而，通过把原来 JSON 里的 symbol 修改为 s，可能仅有 1 字节的区别。而如果使用 CVS 格式代替 JSON，那么使用 event: 可能要多消耗 7 字节。



事件名称可以是除回车和换行之外的任何 Unicode 编码的字符。如果需要多行事件名称，先照照镜子问问自己："真的吗？"如果答案仍然是肯定的，那就要制定一套转义机制，比如，将事件名称转化为 JSON 格式，然后在调用 addEventListener 时使用转化后的事件名称。

在客户端，与之前介绍的使用"信息"处理程序不同的是，这里为每个可能的"事件"创建了处理程序。在这种情况下，那意味着一个外汇对对应一个事件处理程序，如下所示：

```
es.addEventListener("EUR/USD", function (e) {
  var d = JSON.parse(e.data);
```

```
        document.getElementById("EUR/USD").innerHTML = d.bid;
        }, false);

    es.addEventListener("USD/JPY", function (e) {
        var d = JSON.parse(e.data);
        document.getElementById("USD/JPY").innerHTML = d.bid;
        }, false);

    es.addEventListener("AUD/GBP", function (e) {
        var d = JSON.parse(e.data);
        document.getElementById("AUD/GBP").innerHTML = d.bid;
        }, false);
```

我确信这让你打退堂鼓，想要缴械投降了。是的，当多路数据（如本例中的外汇对）以相同的格式、相同的方式处理时，为每条数据流使用 `evnet:` 的成本要比获益多。这不是个好例子，我很抱歉。

有更好的例子吗？有一个每个“事件”的数据都将以不同方式处理的例子。聊天应用如何？可以合理想象一下，数据流是这样发送的：

```
event:enter
data:{id:17653,name:"Sweet Suzy"}

event:message
data:{msg:"Hello everyone!",from:17563}

event:exit
data:1465
```

聊天数据以 JSON 格式发送。JSON 对象有对应实际聊天消息的 `msg` 字段和以 ID 表示发送者的 `from` 字段。当用户进入聊天室时，会触发一个 `enter` 事件，并提供用户 ID 和用户信息（这里只是用户名）。当用户离开聊天室时，会触发一个 `exit` 事件，而数据只是用户的数字形式 ID，而不是一个 JSON 对象：

```
es.addEventListener("enter",
    function(e){ addMember(JSON.parse(e.data)); },false);
es.addEventListener("exit",
    function(e){ removeMember(e.data); },false);
es.addEventListener("message",
    function(e){ addMessage(JSON.parse(e.data)); },false);
```

实际上使用的是下面的函数：

```
function addMember(d) {
    members[d.id] = d;
    var img = document.createElement("img");
    img.id = "member_img_" + d.id;
    img.alt = d.name;
    img.src = "/img/members/" + d.id + ".png";
    document.getElementById("memberimg").appendChild(img);
```

```
    }

function removeMember(id) {
    var img = document.getElementById("member_img_" + id);
    img.parentNode.removeChild(img);
    delete members[id];
    }

function addMessage(d) {
    var msg = document.createElement("div");
    msg.innerHTML = d.msg;
    document.getElementById("messages").appendChild(msg);
    }
```



addMessage() 可能使用了 d.from。这里也省略了错误检测：把这段代码放到实际产品中时需要小心，因为这里没有预防 JavaScript 注入攻击（虽然服务端会处理好从聊天消息中移除问题标签）。

如何使向后兼容方案兼容 evnet: 行？一种方法是在之前介绍过的 processNonSSE(msg) 函数中添加一些代码（参见 6.7.5 节）。这里还需要一个全局变量记录当前处理的事件，如下所示：

```
var currentEvent = null;
...
function processNonSSE(msg) {
var lines = msg.split(/\n/);
for (var ix in lines) {
    var s = lines[ix];
    if (s.length == 0)continue;
    if (s.indexOf("event:") == 0) {
        currentEvent = s.substring(6);
        }
    else{
    if (currentEvent == "exit") {
        removeMember(s);
        }
    else{
        if (s[0] != "{") {
            s = s.substring(s.indexOf("{"));
            if (s.length == 0)continue;
            }
        var d = JSON.parse(s);
        if (currentEvent == "enter")
            addMember(d);
        else if (currentEvent == "message")
            addMessage(d);
        //else unknown event
        }
    }
    }
}
```

注意，这段代码有些复杂，那是因为没有对所有的事件使用 JSON 对象引起的。这也是我建议用 JSON 格式处理所有数据的原因。那会使处理向后兼容方案变得更容易。

上面讲的是第一种方式。另一种方式是给 JSON 对象添加一个 `event` 字段（这仍然需要修改“exit”事件，以便使用 JSON 对象）。这和在 SSE 客户端使用 id: 行的方式相似，但也在 JSON 对象中重复了“id”信息（参见 5.5 节）。

但如果要这么做，何必要麻烦用事件行呢？不如将所有收到的消息放到 `processOneLine(s)` 中，代码有点像下面这样：

```
switch(d){
    case "enter":addMember(d);break;
    case "exit":removeMember(d.id);break;
    case "message":addMessage(d);break;
    }
```

所以，概括来说，SSE 的 event: 特性是一种组织不同行为的方式，但与添加一个额外的 JSON 或 CVS 字段相比，它并没有优势，还不及后者在处理不兼容 SSE 的浏览器时那么容易和高效。所以我建议仅当满足以下两个条件时才使用 `event:`：

- 所有的客户端都原生支持 SSE；
- 想要在事件类型上使用不同的数据类型，包括整型、浮点型或字符串等简单的数据类型（因此不可能包含自己的事件字段）。

## 8.3　多行数据

本书一直提倡对消息使用 JSON 对象格式，其中一个原因是这使我们可以严格地一行一条消息。为什么这样做有好处？因为这使得在旧版浏览器使用的向后兼容方案中进行数据解析变得非常容易。

你是否注意到，在外汇交易应用中，后端如何在 SSE 模式下在数据后面只添加一个额外的回车？在长轮询、XHR 或者 iframe 方案中忽略了这个回车，因为用不上：一行 JSON 总是一条完整的消息（这还顺带节省了 1 字节，或者实际是 6 字节，因为向后兼容方案也没有在数据行加 `data:` 前缀，节省 6 字节并不是这样做的原因。节省客户端处理才是原因所在）。

那么为什么 SSE 需要在消息之间有一个空行呢？这是因为 SSE 标准允许一条消息分割成多行。比如，服务端可以这样发一条消息：

```
data:Roses are red
data:Violets are blue
data:No need to escape
data:When you do as I do
                          <-- Extra LF
```

出于理解客户端如何处理的目的，我们假设服务器每发送一行都会冲刷数据，然后休眠1秒或2秒。客户端会收到“Roses are red”。没有收到空行，所以客户端对它进行缓冲，然后等待。2秒之后又收到了一行，“Violets are blue”，缓冲的数据就是：“Roses are red\nViolets are blue”。注意这只是缓冲，不是告诉客户端有数据到达。在缓冲了第4行之后，完整缓冲是“Roses are red\nViolets are blue\nNo need to escape\nWhen you do as I do”。最后，客户端接收到一个空行，然后调用 JavaScript 事件处理程序，并传递在缓冲中建立的这一行长字符串。



传递给事件处理程序的字符串没有最后的换行。

（SSE 标准解释说，在最后一行数据后面加一个换行会引起客户端的问题，只能删掉最后一个换行。标准就是这样做事，并且常常发现自己在自娱自乐，没有人可以倾述，除了墙角的盆栽。）

如果确实想要在消息的最后加一个空行，该怎么办呢？发送一个空的 data: 行。比如，下面这个数据顺序会给事件处理程序传递“111\n\n333\n\n”字符串：

```
data:111
data:
data:333
data:
data:
```

为什么 SSE 允许这样做呢？这样的话就不需要对回车进行转义了。相反，使用 JSON 的话，上面那段数据就会变成：

```
data:"Roses are red\nViolets are blue\nNo need to escape\nWhen you do as I do"
```

不像本节之前的数据缓冲的例子，\n 占用了2字节，第一个是 \，然后是一个 n。将 data: 和随后的空行计算在内，上面的 JSON 字符串一共是80字节，而非 JSON 格式的91字节。那些 data: 字符串总计所占的字节数，比那些额外的 \ 和两个引号所占的字节数要多。



如何在向后兼容方案中实现处理多行数据的单条消息？基本上需要用 JavaScript 实现本节前面介绍过的 SSE 缓冲算法。并且服务器需要为所有的客户端发送那个额外的空行，不仅是为支持 SSE 的客户端。这没有那么难，并且不需要用 data: 前缀，所以字节数的区别不会有什么问题。但是，和总是使用一行 JSON 的便利性相比，我觉得你需要有一个非常好的理由这样做。

总地来说，使用 SSE 的多行特性，需要同时满足下列条件。

- 所有的客户端都支持原生 SSE。
- 发送的数据原本就是多行的。
- 有一个更好的理由不使用 JSON。

## 8.4 消息中的空白

这一节很短。本书一直使用 `data:XXX`、`event:XXX` 等，SSE 标准也允许写成 `data: XXX`、`event: XXX`。换句话说，可以在冒号后面有空格。我是个随和的人，乐于让人们选择他们自己的处理方式，但这里我要表个态：*绝不要这样做*。因为这样只会浪费字节，并且完全没有任何益处。

但这个特性导致了一个潜在问题：如果把一个未经处理的字符串作为数据，如果数据中需要以一段空格开始，这个空格会被删除。怎么办？简单的方案是使用 JSON。天哪，我确实在反复述说这个，不是么？这个方案有个小小的弊端：每个字符串需要额外的 2 字节（用于引号），如果字符串中有任何特殊字符，还需要额外的转义斜杠字符。但这仍然是个弊端，有其他的解决方案吗？是的，如果需要发送一个未经处理的字符串，并且碰巧这个字符串以一个重要的空格开始，那就在所有的字符串前面加一个空格。这会在每一行浪费 1 字节，如果这种浪费仍然让你烦恼，那就只在数据以空格开始时才这样做……但是为了 1 字节这样做又显得太小题大做了。

## 8.5 又见请求头

在外汇交易应用中，我在 URL 中传入了 `xhr=1` 或 `longpoll=1`，这样服务器就可以识别是哪种向后兼容方案。如果两者都没指定，就默认为 SSE 方案。还有一种方式。在介绍它之前，先回顾一下如何使用这些方案：

`longpoll`

发送一个内容类型为 `text/plain` 的请求头，在发送完一条消息后退出。

`xhr`

发送一个内容类型为 `text/plain` 的请求头。

`sse`

发送一个内容类型为 `text/event-stream` 的请求头。

发送一个 `data:` 前缀和一个额外的回车，以及 `id:` 行。

替代方案是 SSE 客户端发送一个 `Accept: text/event-stream` 请求头，它应该唯一指定客户端原生支持 SSE。所以外汇交易应用原来的代码如下所示：

```
$GLOBALS["is_longpoll"] =i array_key_exists("longpoll", $_POST)
    || array_key_exists("longpoll", $_GET);
$GLOBALS["is_xhr"] = array_key_exists("xhr", $_POST)
    || array_key_exists("xhr", $_GET);
```

```
$GLOBALS["is_sse"] = !($GLOBALS["is_longpoll"] || $GLOBALS["is_xhr"]);

...

if ($GLOBALS["is_sse"]) header("Content-Type: text/event-stream");
else header("Content-Type: text/plain");
```

在使用了前文所说的那种请求头之后，不再需要发送 `xhr=1`，但仍然需要发送 `longpoll=1`，这样它和 `XHR/iframe` 的区别就能识别出来。修改后的代码如下所示：

```
$GLOBALS["is_sse"] = @$_SERVER["HTTP_ACCEPT"] == "text/event-stream";
$GLOBALS["is_longpoll"] = array_key_exists("longpoll", $_POST)
    || array_key_exists("longpoll", $_GET);
$GLOBALS["is_xhr"] = !($GLOBALS["is_longpoll"] || $GLOBALS["is_sse"]);

...

if ($GLOBALS["is_sse"]) header("Content-Type: text/event-stream");
else header("Content-Type: text/plain");
```

你可能已经发现了我没有这样做的原因：相同的复杂度，却没有任何优势。使用显式的 `xhr` 或者 `longpoll` 有两个小优势。首先它可以出现在服务端日志中，而 HTTP 请求头通常不会，这可能有助于故障排查。其次，请求头方案会有一些风险，比如浏览器出 bug 而遗漏发送请求头，或者遗漏了连接符等。而发送 URL 参数完全是无风险的。

## 8.6　这就是全部内容吗

本章介绍了 SSE 的 `event:` 特性，以及它如何支持发送多行数据的消息，以及数据前面的空白会引起问题。我们没有在外汇交易应用中使用这些特性，因为使用了 JSON 之后没有必要用这些特性了。

这就是全部内容吗？不，这仍然不是 SSE 标准的全部内容。还有跨域问题要介绍，这个主题会连同认证授权一起在下一章介绍。

# 第 9 章
# 认证授权：谁在敲门

前面几章中的数据推送应用是对所有人开放的，本章会介绍如何限制访问，不论是通过 IP、Cookie 还是密码。好消息是这就像保护服务器上的其他资源一样直截了当。

但这并不是本章的唯一主题。在之前几章的示例中有另一个潜在的限制，现在也到了处理它的时候了。这个限制就是 HTML 文件（用以发起 SSE 请求和接收数据）和服务端脚本（用以发送数据）必须部署在同一个服务器，好吧，确切地说，必须是同一个域。本章后面会介绍域的定义以及如何应对这种限制。

这两个主题联系紧密，但请注意它们是正交的：可能因为客户端缺少认证授权（IP、Cookie、密码）或来自一个不允许的域，或兼备这两种原因，导致数据推送失败。要使数据推送成功，客户端必须两者都满足。

如果你熟悉网络应用，想要了解本章的内容概要，那么就是认证授权和跨域资源共享几乎就像它们在 Ajax 中的表现一样，但请注意浏览器的支持情况和 bug。

本章结束时，会介绍如何给前面几章中介绍的示例应用添加认证授权以及跨域资源共享支持。

## 9.1 Cookie

Cookie 可以发送给一个 SSE 脚本，在 Cookie 方面，浏览器会像对待其他任何 HTTP 请求一样对待 SSE 连接，不需要额外做什么。下面是一段简单的前端测试代码：

```
<html>
<head>
<title>Cookie logging test</title>
<script>
document.cookie="ssetest=123; path=/";
document.cookie="another-one=123; path=/";
</script>
</head>
<body>
<script>
var es = new EventSource('log_headers.php');
</script>
</body>
</html>
```

当然，这些 Cookie 应该来自网站的另外一个页面，而不是在 JavaScript 中创建。这个例子复用了 8.1 节中的日志脚本。

这个例子也适用于所有的向后兼容方案：`XMLHttpRequests` 和 iframe 请求会被当成其他任何 HTTP 请求处理！

而另一个方向上会怎样呢，SSE 服务端脚本能否将 Cookie 返回呢？答案是肯定的，我们可以测试一下这对脚本。前端没什么可说的，和在第 2 章中见过的 basic_sse.html 无异：

```
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>SSE: access count using cookies</title>
    </head>
    <body>
        <pre id="x">Initializing...</pre>
        <script>
        var es = new EventSource("sse_sending_cookies.php");
        es.addEventListener("message", function (e) {
            document.getElementById("x").innerHTML += "\n" + e.data;
            }, false);
        </script>
    </body>
</html>
```

这个文件就是本书源码的 sse_sending_cookies.html，它连接到 sse_sending_cookies.php，这个文件的代码如下文所示。后端代码看上去与 basic_sse.php 相似，但在文件靠近最上面的部分，会查找名为 `"accessCount"` 的 Cookie（如果没有找到，`@` 会抑制这个错误，并且 `(int)` 转换会将其变为 0），将其增加后返回这个新值，这个新值也显示在输出中：

```
<?php
header("Content-Type: text/event-stream");

$accessCount = (int)@$_COOKIE["accessCount"] + 1;
header("Set-Cookie: accessCount=" . $accessCount);
```

```
while (true) {
    echo "data:" . $accessCount.":".date("Y-m-d H:i:s")."\n\n";
    @ob_flush();@flush();
    sleep(1);
    }
```

运行脚本可以首先看到如下输出：

```
Initializing...
1:2014-02-28 14:17:33
1:2014-02-28 14:17:34
1:2014-02-28 14:17:35
...
```

如果刷新页面，输出就是这样了：

```
Initializing...
2:2014-02-28 14:17:40
2:2014-02-28 14:17:41
2:2014-02-28 14:17:42
...
```

好玩吧！

## 9.2　认证授权（使用Apache服务器）

可以像处理其他 URL 一样对 SSE 脚本进行 IP 限制或密码保护。在本书源码的 .htaccess 文件中，有如下一段代码：

```
<Files "log_headers_ip_restrict.php">
    order deny,allow
    deny from all
    allow from 127.0.0.1
</Files>
```

这是说只允许来自 localhost（127.0.0.1）的浏览器访问，其他任何 IP 的访问都会返回一个 403 错误。可以用 log_headers.ip_restrict.html 测试一下，它只是在尝试连接，并没有做别的（顺便说一下，log_headers_ip_restrict.php 是 log_headers.php 的副本，是在第 8 章创建的，这里创建副本的唯一原因是将这些 IP 地址限制应用到该文件上）。

如果从 127.0.0.1 访问，就会在 tmp.log 上留下记录；如果从其他 IP 访问，则不会在 tmp.log 上留下记录（Apache 甚至都不会启动 PHP 脚本）。浏览器报告这种拒绝访问的方式有所不同，在 Firefox 的 JavaScript 控制台会看到类似“NetworkError: 403 Forbidden - http://example.com/log_headers_ip_restrict.php.”的消息。在 Chrome 中，会在开发者工具的“网络”栏中看到一个被取消的请求。

说句题外话，下面是一段允许所有私有 IPv4 和 IPv6 网络的可选代码块，在许多情况下这

种做法是更实用的：

```
<Files "log_headers_ip_restrict.php">
    order deny,allow
    deny from all
    allow from 127.0.0.1
    allow from 172.16.0.0/12
    allow from 10.0.0.0/8
    allow from 192.168.0.0/16
    allow from fc00::/7
</Files>
```

以上是通过“你是什么”来限制访问，诸如一个 IP 地址。那么通过“你知道什么”来限制访问会怎样呢，比如通过用户名和密码？在 .htaccess 中也有这样的代码块：

```
AuthUserFile /etc/apache2/sse_book_htpasswd
AuthType Basic
AuthName SSEBook
<Files "log_headers_basic_auth.php">
  require valid-user
</Files>
```



事实上在 .htaccess 文件中会发现像下面这样的代码：

```
<Files ~ "^(log_headers_basic_auth[.]php|auth[.]basic_by_apache[.]php)$">
```

因为它也会用在后面小节介绍的脚本中。~ 表示这是一段正则表达式，但这里这个正则表达式仅仅是用竖线隔开的一个二选一的列表。通过将文件名中的 . 转化成字符类型（方括号）进行精确匹配（而不是正则表达式中 . 的含义）。

然后在 /etc/apache2/sse_book_htpasswd 文件中包含下面这些内容：

```
oreilly:AhsbB/t5vHsxA
```

这是一个用于测试的基本认证密码，对应的用户名为“oreilly”。



使用 htpasswd 程序修改密码。密码文件可以放在磁盘的任何位置，不是必须要在 Apache 的配置目录下，只要修改 AuthUser File 来匹配即可。

现在通过浏览器访问 log_headers.basic_auth.html，执行顺序如下。

(1) 加载 log_headers.basic_auth.html，因为它是未受保护的。
(2) 运行 JavaScript，创建 `EventSource` 对象。
(3) 浏览器连接到 log_headers_basic_auth.php，由 Apache 告知需要用户名和密码。
(4) 浏览器弹出一个对话框要求用户输入用户名和密码。

(5) 浏览器再次连接，这次发送了用户名和密码。

(6) Apache 进行验证并运行 PHP 脚本。

(7) PHP 脚本开始向浏览器输出数据（然而在这个例子中，它只将请求头信息记录到日志中，并没有输出任何数据）。

注意，如何认证授权完全是由 Apache 处理的，PHP 脚本不需要做任何事情，并且在认证完成之前不会开始执行。

如果想要 PHP 脚本复查 Apache 是否正确配置并且要求认证授权，可以检查 `REMOTE_USER` 是否设置。log_head ers_basic_auth.php 最上面有这一行：

```
if(!@$_SERVER["REMOTE_USER"])exit;
```

这就像一个冷酷的硬汉保镖在把守，没有密码？休想进来。



PHP 中，可以通过 `$_SERVER["REMOTE_USER"]` 或 `$_SERVER["PHP_AUTH_USER"]` 来获取连接者的用户名。`$_SERVER["PHP_AUTH_PW"]` 是密码（明文）。但如果 PHP 在安全模式下运行，则不能获取到 `PHP_AUTH_*` 的值。

## 9.3　带有SSE的HTTP POST

如果你买这本书只是为了学习如何 POST 变量到 SSE 后端，而且你已经直接跳到这一节，我建议你先来个深呼吸，并确保你现在正在坐着。你看，我有一些坏消息，在想怎么委婉地跟你说……还记得小时候你想像孙悟空一样 72 变，大家都说你没法做到，而你也从来都没有实现，这就证明他们是对的吗？呃，现在这种事又发生了。

SSE 标准不允许 POST 数据到服务端。这是个令人烦恼的疏漏，应该在某个时间点纠正。毕竟，`XMLHttpRequest` 对象允许发送 POST 数据（讽刺的是，这意味着在第 6 章和第 7 章介绍的向后兼容方案中可以轻易地发送 POST 数据）。

在介绍认证授权的本章涉及这方面内容，因为它在做用户登录时会特别令人烦恼。我们不想通过 GET 方式发送用户名和密码，因为那样会在 URL 中可见，并最终记录在一个服务器日志文件中，等等。

SSE 标准也不允许指定 HTTP 请求头，所以用自定义请求头也不行，那又该怎么做呢？

幸好，有一种发送非 URL 数据给 SSE 进程的方法，那就是本章开始所介绍的：Cookie！所以，在你的 JavaScript 中，只需要调用 `new EventSource()` 之前，像这样设置一个 Cookie：

```
document.cookie = "login=oreilly,test;path=/";
```

（我知道你已经意识到需要使它更动态，而不是硬编码的用户名和密码）密码在 Cookie 中是明文的。强烈推荐只在同时使用 SSL 的时候这样用。

然后，是在服务端如何在 PHP 中处理 Cookie 数据：

```
<?php
if (!defined("PASSWORD_DEFAULT")) { // 兼容 5.4.x 以及更早版本
    function password_verify($password, $hash){
    return crypt($password, $hash) === $hash;
    }
}//if (!defined("PASSWORD_DEFAULT")) 结束

$SSE = (@$_SERVER["HTTP_ACCEPT"] == "text/event-stream");
if ($SSE) header("Content-Type: text/event-stream");
else header("Content-Type: text/plain");

if (!array_key_exists("login", $_COOKIE)) {
    echo "data: The login cookie is missing. Exiting.\n\n";
    exit;
    }
list($user, $pw) = explode(",", $_COOKIE["login"]);

$fromDB = '$2a$10$4LLeBta770Y0Z7795j.8' .
    'He/ZCQonnvImXIX0egalzE1MuWiEa6PQa';

if (!password_verify($pw, $fromDB)) {
    echo "data: The login cookie is bad. Exiting.\n\n";
    exit;
    }

while (true) {
    echo "data:" . date("Y-m-d H:i:s") . "\n\n";
    @ob_flush();@flush();
    sleep(1);
    }
```

（这是 auth.custom.php 的全部代码，下一节将会用到这个文件。）

上面的代码首先设置了 SSE 请求头，这样登录错误消息能像其他数据一样发送。然后用 `explode()` 将 CVS 字符串（这里的 Cookie）转化成数组，用 `list($user, $pw)` 将数组转化为两个变量。这里的 `$fromDB` 是一个硬编码的字符串，但是，就像变量名的字面意思所示，它一般应该是一个用以获取散列密码的 SQL 查询。然后密码被散列化并且使用 `password_verify()` 验证，如果与数据库中查询到的不匹配，访问就会被拒绝。



前文代码中硬编码的密码是通过 `password_hash()` 生成的，它和 `password_verify()` 都是在 PHP 5.5 中新增来支持密码安全的函数，它们在早期的 PHP 版本中很容易实现，相关代码可以在附录 C 的 C.5 节中找到（因此前文的代码片段可以用在早期版本的 PHP 中，已经内联定义了 `password_verify()`）。

顺便说一下，网站的任何页面都会接收到登录 Cookie，因为路径指定为 / 了。但是我们可能不希望这样，一旦发生了这种情况，你就需要在产品系统中让 SSE 服务端 URL 看起来像一个路径（比如，使用 Apache 的 `mod_rewrite`）然后将其设置成 Cookie 的路径。

还有，这里设置 `document` 的 Cookie，意味着它关联着加载 HTML 文件所在的域名。在本章后面要介绍的跨域问题中，这意味着如果要链接一个不同的后端，就不能向该后端发送 Cookie 了。所以这个“Cookie 代替 POST”方案只有当 HTML 文件和 SSE 后端在同一个服务器上时才有效。

## 9.4　多重鉴权选择

接下来的示例文件 auth_test.html 给用户提供了 3 种登录网站的方式。第一种是通过 HTML 表单赋值（为方便测试我已经填写好了，但是请不要在产品中这样做）。这种方式是把这些值放在 Cookie 中并提交到前面介绍过的 auth.custom.php 脚本。另外两个按钮会使用 HTTP 基本认证。第一个使用 Apache 认证，第二个使用 PHP 认证。上文已经介绍过 Apache 认证是如何工作的，即通过 .htaccess 文件来控制。

直接在 PHP 中完成基本认证的方式有点像前面小节介绍过的 Cookie 例子，但是不同的是我们将要从 `PHP_AUTH_USER` 和 `PHP_AUTH_PW` 中获取登录细节。下面是从 auth.basic_by_php.php 文件中提取的处理认证的代码片段：

```
$user = @$_SERVER["PHP_AUTH_USER"];
$pw = @$_SERVER["PHP_AUTH_PW"];

$fromDB = '$2a$10$4LLeBta770Y0Z7795j.8' .
    'He/ZCQonnvImXIX0egalzE1MuWiEa6PQa';
if (!password_verify($pw, $fromDB)) {
    header('WWW-Authenticate: Basic realm="SSE Book"');
    header("HTTP/1.0 401 Unauthorized");
    echo "Please authenticate.\n";
    exit;
    }
```

当认证失败时，那些 HTTP 请求头会返回给浏览器，并且会使浏览器弹出一个登录对话框。

下面是完整的 auth_test.html 代码。这是一项有趣的研究，因为它也介绍了如何在需要时创建一个延时的 `EventSource` 连接。相比较而言，实际上之前的例子里都在第一次加载时自动建立了连接。

```
<!doctype html>
<html>
    <head>
        <title>SSE: Basic/Custom Auth Test</title>
        <meta charset="UTF-8">
        <script>
            var es = null;
```

```
        function formSubmit(form) {
            document.cookie = "login="
                + form.username.value
                + "," + form.password.value
                + "; path=/";
            startSSE("auth.custom.php");
            }

        function authByApache() {
            startSSE("auth.basic_by_apache.php");
            }
        function authByPHP() {
            startSSE("auth.basic_by_php.php");
        }

        function startSSE(url) {
            document.getElementById("x").innerHTML = "";
            if (es) {
                document.getElementById("x").innerHTML
                        += "Closing connection.\n";
                es.close();
            }
            document.getElementById("x").innerHTML
                += "Connecting to " + url + "\n";
            es = new EventSource(url);
            es.addEventListener("message", function (e) {
                document.getElementById("x").innerHTML += "\n" + e.data;
            }, false);
        }
    </script>
</head>
<body>
    <div style="float:right">
    <form action="" onSubmit="formSubmit(this);return false">
        Username: <input type="text" name="username" id="username" value="oreilly"/>
        <br/>
        Password: <input type="password" name="password" value="test"/>
        <br/>
        <input type="submit" value="Submit these credentials to auth.custom.php"/>
    </form>
    <br/>
    <button onClick="authByApache()">Use auth.basic_by_apache.php</button>
    <br/>
    <button onClick="authByPHP()">Use auth.basic_by_php.php</button>
</div>
<pre id="x">Waiting...</pre>
</body>
</html>
```

## 9.5　SSL和CORS（连接到其他服务器）

首先是一个好消息，HTTP 和 HTTPS 服务器都可以使用 SSE（以及本书介绍的所有向后兼容方案）。当 HTML 文件从一个 HTTP 服务器下载时，它要连接一个 HTTP 服务器获取数据；当从一个 HTTPS 服务器下载时，它要从一个 HTTPS 服务器 [1] 上获取数据。

注 1：写本书的时候，在 Chrome 中，EventSource 和任何向后兼容方案都不支持自签名 SSL。

如果试图将从一个 HTTP 服务器下载的页面连接到 HTTPS 服务器，或者反过来将从一个 HTTPS 服务器下载的页面连接到 HTTP 服务器的话，Firefox 中会出现“The connection to ... was interrupted while the page was loading.”错误。Chrome 报错消息勉强好一点：“Uncaught Error: SecurityError: DOM Exception 18.”。其他浏览器也都会报一些同样晦涩的消息。事实上，它们都在报关于 CORS 失败的错误。继续往下读。



如果你打算用 Chrome 或者 Safari 来测试接下来几节的代码，确保浏览器版本至少为 Chromium 26 或者 Safari 7，因为在之前的版本中对于 CORS 支持是有缺失或 bug 的。而 Firefox 在很早的版本就已经能够很好地支持了。参见 9.9 节。

CORS 代表 Cross-Origin Resource Sharing（跨域资源共享）。我在想他们是不是先找了一个朗朗上口的缩写然后来找合适的单词。不论如何，CORS 是同域策略的解决方案。同域策略是一个安全特性：如果从一个服务器下载一个 HTML 文件，浏览器只允许连接回那个完全相同的服务器（这并不是专门针对 SSE 的，它也会影响 Ajax 连接和网络字体请求）。

这有些遗憾，对吧？如果 AcmeFeeds 想要在 weather.example.com 域下卖一个天气数据服务，并且希望客户们可以在他们各自的网站上放一些可以连接到到 weather.example.com 的 JavaScript 组件，情况会如何？同域策略不允许这样。

这里有另外一个观点。如果 AcmeWeather 在 weather.example.com 域下有一个天气数据服务，同时它运作一个网站，也在 weather.example.com 域下，通过广告来支付维护数据服务的费用，这样如何？不过 AcmeWeather 不想让其他一些卑鄙的网站偷取它的数据服务，因为那会使它没有广告收入。

浏览器的默认状态是保护 AcmeWeather，浏览器不允许一些人使用其他网站的数据。因此 CORS 的发明是用来允许 AcmeFeeds 覆盖那个默认设定，并告诉全世界它愿意别人来拿它的数据。

基本上，CORS 是一种让服务器放宽同域策略的方式。如果你已经通过 `XMLHttpRequest` 对象（比如，通过 Ajax）使用过 CORS，相信下面这句话会让你开心：`EventSource` 对象基本上以同样的方式工作。

所以，一个域到底是什么？如果满足以下条件则认为两份资源在同一个域。

- 它们的域名匹配（比如，example.com 和 somethingelse.com 是不同的，www1.example.com 和 www2.example.com 是不同的，10.1.2.3 和 example.com 是不同的，即便“example.com”是解析到 10.1.2.3 的）。
- 它们的协议匹配（比如，都是 http:// 或者都是 https://）。

- 它们的端口号匹配（比如，http://example.com:80 和 http://example.com:8080 是不同域，但是 http://example.com 和 http://example.com:80 是同一个域）。

更严格的定义，请参见 http://tools.ietf.org/html/rfc6454#section-4。关于 CORS 全部细节，参见 http://www.w3.org/TR/cors/。

下一节将介绍 CORS 是由 SSE 服务端脚本实现的，而 SSE 服务器端的脚本通过返回额外请求头来表明它允许哪些资源共享。

## 9.6 Allow-Origin

把下面这一行添加到服务端脚本的起始行，来测试一下：

```
header("Access-Control-Allow-Origin: *");
```

这一行是说："任何人从任何地方访问都允许接收这个服务端脚本的数据"。这一段已经添加到 fx_server.cors.php 文件，不过这个文件也只是第 7 章末尾介绍的示例外汇交易应用的服务端脚本的一个副本罢了。下面的附注介绍了如何测试这个请求头，看看它如何正在产生我们所期望效果的。

### 测试 CORS

与之前的例子相比，这个测试需要一些额外的安装工作。需要在一个域下运行 HTML，然后使它连接到另一个域下的 SSE 服务器，即不同的域名、不同的端口或者不同的协议。不过这并不需要两台机器，只需要配置一下服务器。如果不知道怎么做，可以从网上搜到很多与你所使用操作系统和服务器软件相关的配置教程。

为了测试 CORS，这里创建了 fx_client.cors.html 文件，它连接到 fx_server.cors.php 文件。但是，fx_client.cors.html 是本书源码中为数不多的需要在特定条件下才能运行的文件，它取决于你所使用的服务器设置。你不会看到下面这一行代码：

```
var url = "fx_server.cors.php?";
```

取而代之的是：

```
var url = window.location.href.replace(
    "fx_client.cors.html","fx_server.cors.php?");
```

这是设置了一个 URL 绝对地址，而不是相对地址。所以如果 fx_client.cors.html 的访问 URL 是 http://www.example.com/oreilly/sse/listings/fx_client.cors.html，那么 url 就被设置成 http:// www.example.com/oreilly/sse/listings/**fx_server.cors.php?**。

接下来看：

```
if(url.indexOf("https") >= 0)
    url = url.replace("https://","http://");
else url = url.replace("http://","https://");
```

这段代码用来在 HTTP 和 HTTPS 之间转换。为了支持这段代码的功能实现，这里设置 Apache SSL 为一个自签名证书，但是在同一个 IP 地址并指向同一个 DocumentRoot。所以当浏览 http://www.example.com/oreilly/sse/listings/fx_client.cors.html 时，它连接到 **https:**//www.example.com/oreilly/sse/listings/fx_server.cors.php?，而当浏览 https://www.example.com/oreilly/sse/listings/fx_client.cors.html 时，它连接到 **http:**//www.example.com/oreilly/sse/listings/fx_server.cors.php?。

接下来是在主机名部分不同的测试域名方法：

```
url = url.replace("//www1.","//www.");
```

当浏览 www1.example.com 时，它会连接到 www.example.com，当浏览 www.example.com，或其他非 www1 的地址时，它什么都不做，同时也会继续连接到这个与第一次连接时相同的域。

这里通过复制虚拟主机 www.example.com 配置（包括 HTTP 和 HTTPS）和将其改名为 example1.com 的方式来配置 Apache 以便能够处理上面所有的情况。因此，当浏览 http://www1.example.com/oreilly/sse/listings/fx_client.cors.html，它连接到 https://**www**.example.com/oreilly/sse/listings/fx_server.cors.php?。

测试的时候，新增一个 IP 地址总是比新增一个主机名称要容易，下面是一种转化 IP 地址为 URL 的方法：

```
url = url.replace(
    /([/][/]\d+[.]\d+[.]\d+)[.]51[/]/,
    "$1.50/");
```

万恶的正则表达式啊！我们简单来说，这里是把 IP 地址最后一段的“51”变成了“50”，所以如果浏览 http://10.0.0.51/oreilly/sse/listings/fx_client.cors.html，就会连接到 **https:// 10.0.0.50**/oreilly/sse/listings/fx_server.cors.php?。

最后，添加一段代码来报告所发生的变化；当然这只是为了排查故障：

```
console.log("Our URL is "
    + window.location.href
    + "; connecting to " + url);
```

现在，来验证一下这些修改是否真的有效。首先用两个不同的域名，分别以 http:// 和 https:// 来浏览 fx_server.cors.html。这应该是有效的，然后编辑 fx_server.cors.php 来向 `header("Access-Control-Allow-Origin: *");` 添加注释；这时候所有的变量应该都失效了。

## 9.7 完善访问控制

`header("Access-Control-Allow-Origin: *");` 中的 * 使它向任何人（张三、李四、王五，等等）开放。幸好，完善访问控制是完全可行的。比如，像这样修改：`header("Access-Control-Allow-Origin: http://www.example.com");`（http:// 前缀必须要有）。现在，浏览 http://www.example.com/oreilly/sse/listings/fx_client.cors.html，它连接到 https://www.example.com/oreilly/sse/listings/fx_server.cors.php?，成功了。但是，就如本章开始所介绍的那样，下面任何一个访问都会失败：

- **https**://www.example.com/.../fx_client.cors.html
- http://**www1**.example.com/.../fx_client.cors.html
- http://www.example.com:**88**/.../fx_client.cors.html
- http://**some.other.domain.com**/.../fx_client.cors.html



`Access-Control-Allow-Origin` 并不能代替合理的认证，因为客户端可以伪造 Origin 请求头。同时还要记得它还依赖于浏览器是否正确地实现了 CORS。

CORS 没有想象得那么灵活。可以用“*”表示完全开放，或者明确地指定一个域，比如，一个 HTTP/HTTPS、域名、端口号的组合。两种选择：一个域或所有域，任何这两者之间的情况，需要在脚本中解析 Origin 请求头。下面是一个最基本的例子，实际上与使用 `"Access-Control-Allow-Origin: *"` 完全相同[2]：

```
header("Access-Control-Allow-Origin: ".@$_SERVER[‘HTTP_ORIGIN']);
```

（@ 符号意思是抑制错误，所以如果没有设置 `HTTP_ORIGIN`，它会静默地返回一个空字符串，这种情况意味着 CORS 会总是拒绝连接。）

下面是一个更有趣的例子：

```
if(preg_match(‘|https?://www[1-6]\.example\.com$|',@$_SERVER["HTTP_ORIGIN"]))
    header("Access-Control-Allow-Origin: ".$_SERVER["HTTP_ORIGIN"]);
else header("Access-Control-Allow-Origin: http://www.example.com");
```

最后一行是说，如果正则表达式没有匹配成功，就需要从 http://www.example.com 浏览。9.12.3 节的附注栏“比较两个 URL”中会介绍正则表达式，但这里相对简单：它是说任何 wwwN.example.com 都可以匹配（N 是 1、2、3、4、5 或 6），然后会显示可以连接。这里也显式地允许 HTTP 和 HTTPS 两种类型的 URL（“s”后面的问号意思是“s”是可选的）。

注 2：我是说完全相同吗？当使用证书时有一个关键的不同，请参见 9.10 节“构造函数与证书”。

注意 https://www.example.com 会匹配失败，因为它不是 www1 到 www6 之间的任何一个，在 [1-6] 后面加一个 ? 也可以使它是可选的。

没觉得我们可以做得比这更好一点么？我们的目的是只有从 www.example.com、www1.example.com 等域下载应用 HTML 的客户端允许连接，其他的都不行，那要是把这个目的写得更显式一点呢：

```
if(preg_match('|https?://www[1-6]?\.example\.com$|',@$_SERVER["HTTP_ORIGIN"]))
    header("Access-Control-Allow-Origin: ".$_SERVER["HTTP_ORIGIN"]);
else{
    header("HTTP/1.1 403 Forbidden");
    exit;
    }
```

这里调整了一下正则表达式以是它可以覆盖到 www.example.com（以及这个子域的 HTTP 和 HTTPS 地址），但任何其他的域试图连接，都会立即死掉。这段代码会放到服务端脚本的最上面。

## 9.8 HEAD和OPTIONS

到目前为止，我们只考虑了浏览器发送 GET 或 POST 请求的可能性。当请求一个新数据流时发送一些其他的请求（比如 PUT）会显得很奇怪。在 PHP 中，至少所有的请求方式都是同等对待的。如果客户端发送一个 HEAD 请求，现在的代码会表现得很糟糕：我们没有事先假设发送任何请求体，而事实上这里不仅仅是发送内容，并且会一直保持连接。解决这个问题的一种选择是刚好在进入主循环之前（比如在所有的请求发送之后）就返回 HEAD 请求。而另一种方式是将 HEAD 请求定性为不合理的，并拒绝接收请求。如果想要实现这个方法，就可以将下面的代码放到脚本的顶部：

```
switch ($_SERVER["REQUEST_METHOD"]) {
    case "GET":case "POST":break;
    case "OPTIONS":break; //TODO
    default:
        header("HTTP/1.0 405 Method Not Allowed");
        header("Allow: GET,POST,OPTIONS");
        exit;
        }
    }
```



这个 HTTP 方法检测实际上与本章认证授权和 CORS 的主题无关。现在开始讨论是为介绍 OPTIONS 的处理方法（马上开始）做铺垫。

如果你有这个需求，那么对于前面几章那些只对 GET 请求有效的例子，只需将下面这段代码放到脚本的顶部即可：

```
if($_SERVER['REQUEST_METHOD']!='GET') ↵
{header("HTTP/1.0 405 Method Not Allowed");header("Allow: GET");exit;}
```

当返回 405 时，Allow 请求头是必须的，它指明了什么请求头是允许的，如果它指定为 `OPTIONS` 会怎样？

这个方案是浏览器可以用一个 `OPTIONS` 方法调用脚本，来获取支持的 HTTP 协议组件信息。在 CORS 的话题范围内，称为预检请求，通常用来询问什么信息可以发送给上述的域。



如果用 Apache 认证，注意 `OPTIONS` 请求会以返回 401（“要求认证”）的形式失败，并且永远不会实际访问到脚本。浏览器一般会提示用户进行认证，但也有些浏览器（比如，Safari 5.1）不会。

让人头疼的是，给 `"Access-Control-Allow-Headers"` 响应返回一个通配符似乎并不能起作用，所以不得不浪费带宽来试图猜测浏览器可能发送的每一种请求头。下面是一种实现的方法：

```
...
case "OPTIONS":
    header("Access-Control-Allow-Origin: *");
    header("Access-Control-Allow-Headers: Last-Event-ID,".
      " Origin, X-Requested-With, Content-Type, Accept,".
      " Authorization");
    exit;
```

如果用 Node.js 做同样的事，方案其实是类似的。可是，用 Node.js 处理 POST 有一点复杂，所以用了两个专用的函数（这里没有介绍）分别处理 GET 和 POST，而请求处理函数完全被下面这个切换函数取代：

```
function (request, response) {
    switch (request.method) {
        case "GET":
            handleGET(request, response);
            break;
        case "POST":
            handlePOST(request, response);
            break;
        case "OPTIONS":
            response.writeHead(200, {
                "Access-Control-Allow-Origin: *",
                "Access-Control-Allow-Headers: Last-Event-ID," +
                    " Origin, X-Requested-With, Content-Type, Accept," +
                    " Authorization"
            });
            break;
        default:
            response.writeHead(405, {
```

```
            "Allow: GET,POST,OPTIONS"
        });
        break;
    }
}
```

## 9.9 Chrome和Safari以及CORS

基于 Webkit 的浏览器已经有一些妨碍 CORS 在原生 SSE 中正常运行的 bug。在 Chrome 25 以及更早版本和 Safari 6 以及更早版本中，`EventSource` 的 CORS 实现是被破坏 / 缺失的。在我敲下这些文字时，许多人已经不再使用 Chrome 25 了，但相当一部分人还在使用 Safari 6。如果这算是个坏消息，那么更坏的消息是这个几乎不可能用特性检测。

如果 CORS 是你系统必要的部分，那么替代方案就是强制 Chrome 和 Safari 使用 XHR 来取代 SSE。这听起来很可怕，不是么？不过，这其实并没有那么糟，因为在带宽和连接方面，XHR 做的几乎和 SSE 一样好。事实上 XHR 与 SSE 相比只有两个缺点。

- 需要额外的代码使之同时支持 SSE 和 XHR，但这个我们已经做了。
- 当内存占用太多时需要重连，参见 7.5 节。

本章末尾的示例中，使用了浏览器版本检测通知旧版的 Chrome 和 Safari 用 XHR 取代原生 SSE。因为它只是做了一些正则匹配，所以本书没有介绍，有兴趣的话，可以看一下本书源码 fx_client.auth.html 中的 `function oldSafariChromeDetect()`。

Chrome 有另一个问题，虽然只是在开发和测试时才有：自签名的 SSL 证书会被拒绝。XHR 和 SSE 都有这个问题，用命令行标记 `--disable-web-security` 也不起作用。所以，这不是 Chrome 的 SSE 实现所特有的问题。事实上，这个 bug 甚至不是 CORS 特有的：不能用 `XMLHttpRequest` 或 `EventSource` 连接到一个自签名 HTTPS 服务器，就是这样[3]。你可以通过将服务器证书添加为本地信任根证书的方式解决这个问题，或是等浏览器开发者解决。或者，因为自签名证书一般只在测试和开发时使用，可以用 Firefox 和其他浏览器开发，然后只在产品服务器上用 Chrome 测试。

iOS 7 可以在原生 SSE 上运行 CORS，但是，当连接到一个请求认证的 SSE 数据源时，这时却无法弹出提示密码的对话框。XHR 也有同样的问题，所以没法解决。如果想要支持 iPhone/iPad，则需要安排用户直接访问目标服务器的页面，以便他们可以获得提示输入用户名密码的弹框。浏览器会等待那些证书，而这些证书会在 SSE 或 Ajax 建立连接时发出（Cookie 认证方式没有这个问题）。

注 3：可以在 http://code.google.com/p/chromium/issues/detail?id=96007 关注一下这个 bug 报告。

## 9.10 构造函数与证书

现在你肯定知道 EventSource 构造函数利用 URL 参数进行连接了。它还有第二个参数，这是一个带有选项的对象。目前为止，只有一个可能的选项，withCredentials，它是布尔类型，默认值为 false。

试一下把前面示例中的这个值都设为 true，把下面这一行：

```
es = new EventSource(u);
```

修改为：

```
es = new EventSource(u, { withCredentials: true } );
```

如果连接到的是同一个服务器，这不会有什么影响。但是在 CORS 示例中做这个修改，尝试连接到一个不同的协议、主机或端口。它就会中断。在 Firefox 浏览器上会报出“连接中断”异常。

为了解决这个问题，可以让服务端脚本发送下面两个请求头，而不是 header("Access-Control-Allow-Origin: *");：

```
header("Access-Control-Allow-Origin: ".@$_SERVER["HTTP_ORIGIN"]);
header("Access-Control-Allow-Credentials: true");
```

上面第二个请求头是说：“是的，我们很乐意由你来发送证书”。但这些证书不允许和 "Access-Control-Allow-Origin: *" 一起使用。浏览器会觉得这有点太混杂。所以，还记得前面提到过的那句和 "Access-Control-Allow-Origin: *" 等效的代码吗？放到这里就完美了。它事实上做着同样的事，但浏览器会运行。“啊噢，服务器显然已经听从了安全沟通的演讲，所以当它说它想允许证书时，我们也相信它吧。”

站起来跳一段欢快的舞蹈吧，因为有了那两行代码，客户端就能添加 withCredentials:true 配置了，并且一切又重新奏效了。但是我们刚刚允许发生的事情到底是什么 ?!

## 9.11 withCredentials

假设从 http://example.com/index.html 下载 HTML 文件，它试图向 http://www1.example.com/sse.php 建立 SSE 连接。如果到本章目前为止这样做过，那么你一定知道这会因为同域策略而失败。并且你知道通过使服务端设置 "Access-Control-Allow-Origin:" 为 *，或是你客户端的域，都可以覆盖同域策略，同时连接也可以正常工作。

你也知道，如果已经这么做，HTTP 认证也可以在 SSE 下正常运行。

不过如果试图将这两件事合并，问题就产生了。默认情况下，当访问另一个域时，不会发

送认证所需的请求头。如果 SSE 服务端脚本（或者 Apache）返回一个 401（一般这会使浏览器弹出对话框要求用户输入用户名和密码），它会被当成错误处理。



前面介绍过用 Cookie 实现的自定义登录系统，在 `EventSource.withCredentials` 不支持 POST 的场合，也表明可以发送 Cookie，但在这里没用，因为我们只能在文档上设置 Cookie，这意味着我们是在我们的域上设置，而在一个 IP 地址或主机名上注册的 Cookie 不能发送到另一个不同的 IP 地址或主机名上。

那又是什么意思？意思是不能在自定义登录系统中使用原生 SSE 来连接一个不同的域。简单地说就是不可能[4]。

那解决方案呢？希望 SSE 标准将来的版本可以允许 POST 数据。特殊处理呢？在本章末尾的例子中，当检测出是在试图做一个跨域的自定义认证，会强制使用 XHR 连接替代 SSE。其他方法会有使用基本的认证，避免使用不同的域，或者使用越界认证，因此发送给 SSE 服务端的 Cookie 就能在打开 SSE 连接之前被接收到。

因此，要做到这些，客户端需要将 `{ withCredentials: true }` 作为第二个参数传递给 `EventSource` 构造函数，如上节所介绍的那样，服务端需要返回 `"Access-Control-Allow-Credentials"` 请求头，设置为 `true`，同时设置 `"Access-Control-Allow-Origin"` 为客户端指定的任意域。一旦做了这些，HTTP 认证（以及 Cookie[5]）就能运作于 CORS 了。

好吧，它们可以在现代浏览器上运行。XHR 以完全一样的方式运作，所以它们也能运行于向后兼容方案。好吧，呃……看下一节吧。



再次提醒，这些都不是真正的安全。它全依赖于客户端遵循规则。用几行你所选的后端语言代码，或者一个 curl 单行小程序，就可以发送认证请求头、Cookie、GET 数据、POST 数据，甚至一张伊丽莎白二世的照片给任何 SSE 服务器，不论它们是否返回 `Access-Control-` 请求头。不仅如此，你还可以伪造 `User-Agent` 和 `Origin` 请求头。

CORS，以及 withCredentials，主要用以防止跨站请求伪造（CSRF）和类似攻击。

注 4：好吧，不完全是。在 Firefox，只是你可以从 http://example.com 发送 Cookie 到 https://example.com，反之亦然（即只在协议部分不同的域）。我的建议是不要依赖这个特性，因为这不符合 XHR 的 CORS/Cookie 特性，所以可能将来会修改。

注 5：指允许跨域发送的，其他的仍然试用这条规则，比如，www1.example.com 的 Cookie 不能发送到 www2.example.com。

## 9.12 CORS和向后兼容方案

本书通篇都在介绍能在旧版浏览器上运行的 SSE 等效方案，我已经尝试达到了 99% 的浏览器覆盖率。好消息是 CORS 在 XHR[6] 中是可用的并且完全以相同的方式运作。因为它们使用 `XMLHttpRequest`，因此不需要为第 6 章和第 7 章介绍的长轮询或 XHR 技术做任何修改。但是 IE9 及其更早版本有一些问题。

但是在看 IE8/IE9 之前，先给支持 XHR 的浏览器添加 CORS 支持。是的，那已经做完了，又快又简单的原因是 CORS 随着服务端请求头一起完全做完了，而 JavaScript API 没有任何修改。

但那是不支持证书的 CORS。比如，示例应用发送一个自定义请求头（`Last-Event-ID`）。因此，必须使用 `withCredentials`，而不是纯粹的 CORS。现在来给支持 XHR 的浏览器添加 `withCredentials`，这需要修改 `startXHR()` 函数：

```
function startXHR() {
    ...
    xhr = new XMLHttpRequest();
    ...
    xhr.open("GET", u, true);
    if (lastId)xhr.setRequestHeader("Last-Event-ID", lastId);
    xhr.send(null);
}
```

与 SSE `EventSource` 构造函数的 `options` 对象不同的是，XHR 设置第三个参数为 `true`。

接下来，在 `startLongPoll` 函数中做同样的修改：

```
function startLongPoll() {
    ...
    if (window.XMLHttpRequest)xhr = new XMLHttpRequest();
    else {
        document.getElementById("msg").innerHTML +=
            "** Your browser does not support XMLHttpRequest. Sorry.**<br>";
    }
    ...
    if ("withCredentials" in xhr) {
        xhr.open("GET", u, true);
    } else {
        document.getElementById("msg").innerHTML +=
            "** Your browser does not support CORS. Sorry.**<br>";
    }
    if (lastId)xhr.setRequestHeader("Last-Event-ID", lastId);
    xhr.send(null);
}
```

注 6：Firefox 自从 3.5 就支持 XHR 的 CORS，Chrome 从 4.0 开始，Safari 从 4.0 开始，IE 从 8.0 或者 10.0，这取决于所支持的程度，iOS Safari 和 Andorid 内置浏览器分别从 3.2 和 2.1 开始。换句话说，可以认为除了 IE 之外的所有浏览器都支持 `XMLHttpRequest` 的 CORS。

这里不仅 `open()` 的第三个参数为 `true`，也删除了 IE6/IE7 的 Ajax 相关代码，取而代之的是一个报错信息，这才是 IE6/IE7 要关心的。接下来检测是否支持 `withCredentials`，如果不支持（比如 IE8/IE9），就报一个错误（这是下一节要关心的）。

（可以从本书源码的 fx_client.cors_xhr.html 文件找到上面这段代码。）



我本应该在 `startXHR()` 中也做相同的检测，没有这样做是因为 `connect()` 的代码已经确保 IE9 及其更早版本不会执行到 `startXHR()`。我还没有发现一款以 `startXHR()` 结尾但却不支持的 CORS 和证书的浏览器，如果你发现了，还请告诉我。

## 9.12.1 CORS和IE9及其更早版本

在之前的章节我提到过长轮询和 XHR 技术很好。第 7 章介绍的 iframe 技术是另一回事，它不管用。由于安全原因，一个 iframe 不能访问来自另一个域的 iframe 的内容，也没有类似 CORS 的变通方案可以用。所以，这就意味着 IE8 和 IE9 必须使用长轮询来解决应用中的跨域问题。



如果你说，“IE6 或者 IE7 怎么办”？你问得太多了：它们没有一套我们可以使用的 CORS 机制，即便是使用 XHR（比如长轮询）。所以，简单地说，IE7 及其更早版本不能在跨域情况下运行，且必须将 HTML 和数据推送服务放到同一个域。

是否需要使用 `withCredentials` 呢？问题其实是，是否需要向不同的域服务器发送认证请求头或 Cookie，并且能在 IE8/IE9 上运行？对不起，那个要求太多了。问题是 IE8 和 IE9 的 CORS 等效方案，称之为 `XDomainRequest`，它明确地拒绝发送任何自定义请求头（包括认证请求头）和 Cookie。如果必须认证并且必须支持 IE8/IE9，那就必须将 HTML 页面和 SSE 服务放到同一个域（使用一个负载均衡或者反向代理让所有的服务器都使用同一个域名，然后用其他方式来标识它们之间的不同）。



IE10 及其之后的版本已经使用了 XHR 技术，支持 CORS，并且也支持 `withCredentials`！不需为 IE10 及其之后版本做任何修改。

`XDomainRequest` 比真正的 CORS 限制更多[7]。不论是限制“只能是 GET 或 POST 数据”，还是限制 MIME 类型必须是 `text/plain`，都没什么影响。但需要注意一个不同点：决不允许

注 7：关于 `XDomainRequest` 如何在 IE8 和 IE9 中运行，参见 http://bit.ly/1csbEHT。注意只允许 GET 和 POST 数据，并且必须是 `text/plain`，不允许发送 Cookie。

不同的机制。那意味着一个来自 http://example.com 的 HTML 页面不能访问 https://example.com 上的服务，反之亦然。服务器没办法说它很好。

下面是如何修改 `startLongPoll()` 使用 `XDomainRequest`，从而使 IE8 和 IE9 支持 CORS：

```
if ("withCredentials" in xhr) {
    xhr.open("GET", u, true);
} else if (typeof XDomainRequest != "undefined") {
    xhr = new XDomainRequest();
    xhr.open("GET", u);
} else {
    document.getElementById("msg").innerHTML +=
        "** Your browser does not support CORS. Sorry.**<br>";
}
xhr.onreadystatechange = longPollOnReadyStateChange;
```

如你所见，`XDomainRequest` 是 `XMLHttpRequest` 的一个简单的替代。可是，这种特性检测的方式意味着不能在创建 `XMLHttpRequest` 对象之前确定是否需要它。因为 `xhr` 可能要再创建一次，所以在这个代码块结束之前不能做任何事情。这也是为什么将给 `xhr.onreadystatechange` 的赋值挪到了那段代码块之后。

接下来两节将介绍两种处理 IE9 及其更早版本使用 `startLongPoll()` 的方法。

## 9.12.2　IE8/IE9：总是使用长轮询

如果已经明确总是要处理跨域问题，这就好办了，在 `connect()` 函数中，将下面这一段：

```
...
else if(isIE9OrEarlier){
    if(window.postMessage)startIframe();
    else startLongPoll();
}
...
```

修改为：

```
...
else if(isIE9OrEarlier){
     startLongPoll();
}
...
```

作为奖励，现在可以移除 iframe 相关代码了，那意味着，下面这些可以移除。

- 整个 `function startIframe()` 函数。
- `function disconnect()` 中的两行。
- `var iframe` 和 `var iframeTimer`。

### 9.12.3 动态处理IE9及其更早版本

如果不知道是否会命中安全限制怎么办？这可能因为是一份会在多个网站使用的库代码。也可能它仅仅是一个动态发送到浏览器端的 URL，并不知道会连接到同一个服务器还是不同服务器[8]。在这种情况下，把前面的代码修改成下面这样：

```
...
else if(is_ie_9_or_earlier){
    if(window.postMessage && isSameDomain())
        start_iframe();
    else start_longpoll();
}
...
```

所有额外的逻辑已隐藏到 `isSameDomain()` 函数中[9]。`isSameDomain()` 函数需要做什么？它需要比较 `url` 和 `window.location.href`，并且当下面这几条都相同时返回 `true`：

- 协议（HTTP 和 HTTPS）
- 服务器名称（或者 IP 地址）
- 端口号

有两种方式来写这个功能代码。一个是使用正则表达式，另一种是用一个 JavaScript+DOM 的技巧。下面的附注中介绍了这两种方式（本书源码 fx_client.cors_xhr_ie.html 中实现了这两种方式，但是使用了正则表达式的方案）。

**比较两个 URL**

任何时候，当需要比较两个字符串的多个部分时，正则表达式都是一个很好的工具。如果因为它们看起来非常难读就抗拒学习正则表达式，那就放弃这个想法吧。在你仅仅知道基本的正则语法的时就已经能做很多事情了。

此外，不论你的正则表达式水平如何，可以看一下这个测试工具：http://www.regexplanet.com/advanced/java script/index.html。当你顺着里边的注释往下看的时候，你会觉得它十分有帮助。

下面是从一个 URL 解析协议，服务器名称，和端口号的正则表达式：

```
/^(https?):[/][/]([^/:]+)(:([^/]+))?/
```

注 8：我觉得“一个动态发送到浏览器端的 URL”这句有点夸张，在这种场景下，它听起来似乎多半都会连接到另一个服务器。如果是这样，把代码简化成总是使用长轮询。

注 9：在本章的最后一个例子中，在决定是使用 SSE 和 Cookie，还是回退到 XHR 方案以便可以发送 POST 数据时，这个函数会再出现一次。

两头的 / 标示了正则表达式的开始和结束。^ 意思是这需要在字符串开始的部分匹配。小括号括起来的部分是想要捕获的，这里有 3 块捕获，如下面粗体部分：

```
/^(https?):[/][/]([^/:]+)(:([^/]+))?/
```

第一段捕获字符串是针对协议的（HTTP 或 HTTPS），第二部分是域名，第三部分是可选的端口号。小括号后面的 ? 表示 0 或 1 次，所以如果没有端口号，第 3 个捕获字符串就会是 undefined。第二个块，[^/:]+，是说抓取任何字符直至遇到斜杠或一个冒号（斜杠或冒号不会包含在捕获字符串中）。接下来，[^/]+，是说捕获任何字符直至遇到斜杠。这两种情况中，字符串的结束也会终止捕获（还有一对小括号，它是用来分组的，而不是捕获的。它们的目的是确保分号前缀不是被捕获端口号的一部分）。

在协议和域名之间是 ://。为什么要用这么有趣的符号（:[/][/]）？斜杠符号已经用来标识正则表达式的开始和结束，所以在其他地方用斜杠时需要进行转义。但是，它们**不**需要以字符类型转义，方括号标识字符类型，所以 [/] 和 \/ 是一样的，都表示匹配一个斜杠。我个人认为字符类型方式更清晰（特别是当把正则表达式放到反斜杠需要转义的字符串中：那样斜杠最终看起来就像这样 \\/ 或者甚至这样 \\\\/）。



在 /.../ 之间定义一个正则表达式隐式地创建了一个 RegExp 对象。也可以通过 `var re = new RegExp('^(https?)://([^/:]+)(:([^/]+))?');` 来显式地创建一个对象，这些方式都是一样的。注意第二种方式，“/”不再用来开始和结束正则表达式，所以不再需要转义！所以我可以直接用“/”字符而不用写成 [/]。

也可以不把正则表达式赋值给 re 变量，而是把开始的两行合并成一行：`var m1 = /^(https?):[/] [/]([^/:]+)(:([^/]+))?/.exec( url )`。

有两个原因表明这不是一个好方案，都是要用这个正则表达式两次。第一个原因很明显，重复代码不是件好事。第二个原因是将正则表达式赋值给一个变量时进行了编译。因为我们使用了那个复杂的正则表达式两次，我们为 CPU 节省了一次额外的正则表达式编译的开销，这里那个开销很小，不过如果正则表达式在一个 1000 次的循环里的话，影响就会更大。但是，原则上来说，如果要使用一次以上，应该总是把正则表达式赋值给一个变量。说得深入一点，如果一个正则表达式被调用多次，比如服务器每次发送数据，那么应该尝试将正则表达式赋值给一个全局变量，这样它在整个脚本中只会被编译一次。

前面聊那么多，体现到 JavaScript 中，就是下面这样：

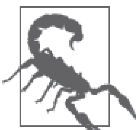
```
function is SameDomain(){

    var re = /^(https?):[/][/]([^/:]+)(:([^/]+))?/;
    var m1 = re.exec(url);
```

```
        if (!m1)return true;
        var m2 = re.exec(window.location.href);
        if (m1[1] != m2[1])return false;
        if (m1[2] != m2[2])return false;
        if (m1[4] != m2[4]) {
            if (!m1[4])m1[4] = (m1[1] == 'http') ? "80" : "443";
            if (!m2[4])m2[4] = (m2[1] == 'http') ? "80" : "443";
            if (m1[4] != m2[4])return false;
            }
          return true;
    }
```

调用 RegExp 对象的 exec 会返回一个匹配数组。[1] 是第一个匹配（协议），[2] 是第二个匹配（服务器名称），[4] 是端口号（[3] 是包含了冒号的端口号，这里没有用到）。端口号需要额外两行代码，因为如果一个包含了默认端口号而另一个没有，它们也应该能匹配。那就是，http://example.com/ 和 http://example.com:80/ 是一样的（如果你发现任何浏览器把它们当成不一样的，就给浏览器发送一个 Bug 报告，然后加一点特殊处理让那个浏览器不执行那两行代码！）。

如果 URL 是相对的，正则表达式会失效。换句话说，URL 不是 http://exam ple.com/fx_server.php，而是 /fx_server.php。事实证明这是一种解决方案，而不是问题：根据定义，相对 URL 一定是同域的！所以，如果正则表达式不匹配，就会假定它是一个相对 URL，并且立即返回 `true`，这就是 `if(!m1)return true`; 这一行所做的事。



这里假定 url 永远没有不好的值，但那应该在应用的控制之下。无论如何，最糟糕的是，由于一个坏的 URL，IE8 会试图使用 iframe 并且失效（出于安全原因），而不是使用长轮询并且失效（因为 URL 是坏的）。



正则表达式遇到"//example.com/..."类型的 URL 也会失效，而这些 URL 要使用相同的协议（允许在 HTTP 和 HTTPS 站点之间共享代码）。我即便是在处理这种情况时也不会让正则表达式变得更复杂，因为使用两到三个易于理解的正则表达式比用一个包含了所有情况的怪兽更好。你要找的正则表达式是 /^([/][/])([^/:] +)(:([^/]+))?/，fx_client.cors_xhr_ie.html 文件有它的完整实现。

我说过还有一种做法，来直接看代码：

```
    function isSameDomain() {
        var m1 = document.createElement("a");
        m1.href = url;
        var m2 = document.createElement("a");
        m2.href = window.location.href;
        if (m1.protocol != m2.protocol)return false;
        if (m1.hostname != m2.hostname)return false;
        if (m1.port != m2.port)return false;
        return true;
    }
```

这依赖于 JavaScript 在 DOM 中创建一个 <a> 标签时得到一个完整的 Location 对象，而这个对象已经把所有这些可爱的字段给你准备好了。眼前看着的不是正则表达式，真酷。不过它的缺点是它有些脆弱，不支持 IE6 或 IE7，还有其他一些细微的浏览器差异。你也需要测试它如何在所有的浏览器中运行，包括前面代码中要处理的各种边角情况（相对 URL，“//example.com/”URL，一个显示包含端口号而另一个没有，等等）。



我是在 https://gist.github.com/jlong/2428561了解到这种技术的，虽然很明显它在那之前就已经被发现了，但是那个页面中的评论也很有学习价值。

## 9.13 汇总

最后两节是否让你大头嗡嗡，两眼汪汪，并且开始觉得去山里放羊都是一个不错的职业选择？ IE 就是就这个能力。好吧，好消息是你差不多已经读完这本书了，还有几页就到附录了。但在我们分道扬镳之前，我们还需要做一个例子：让我们（开个玩笑）把本章前面介绍的示例应用的 CORS 版本并入 auth.html 示例。因此，数据流会在登录之后才开始，登录可以通过基本认证或 Cookie 实现。让我们（开一些严肃的玩笑[10]）使它也支持所有的目标浏览器。好吧，就如已经解释的那样，那意味着不能支持 IE8 和 IE9：它们的 CORS 实现在设计上与进行认证不兼容（页面可以在 IE8 上运行，但当把目标 URL 设置为一个不同域时就会中断）。可是，fx_client.auth.html 确实检测了 Chromium 25 及其更早版本，Safari 及其更早版本，强制它们使用 XHR 取代原生 SSE，以支持 CORS。

这意味着这个例子中只有这些浏览器使用原生 SSE：Firefox 10+、Opera 12+、Chrome 26+、Safari 7+。并且，当使用“自定义”登录技术并涉及跨域时，所有的浏览器都会回退到使用 XHR 技术（因为 XHR 能使用 POST，而 SSE 只能用 Cookie，而 Cookie 不能跨域）。

### 后端文件

这个例子比它实际所需的要复杂，因为它支持本章前面介绍的三到四种不同的认证方式，不过要把所有的代码放到两个文件中。首先是 fx_server.auth.inc1.php（设置了一些全局变量，定义了所有的类和函数），然后是 fx_server.auth.inc2.php，做了剩下的全局代码和主循环。fx_server.auth.inc1.php 和 fx_server.auth.inc2.php 的代码基本上来自第 7 章末尾的 fx_server.xhr.php，把它分成了两部分，一些代码移到了专门的认证文件里。

其他四个文件（fx_server.auth.apache.php、fx_server.auth.php.php、fx_server.auth.cus

注 10：如果你是漂亮的，女性，并且实际上并不认为那听起来有趣，我们应该在一起……不，等等，肯定有陷阱，没有人会那么完美。你可能有一些怪异的嗜好，包括蟾蜍或 Excel 或某些东西。

tom.php 和 fx_server.auth.noauth.php）做了它们各自专门的认证代码，一个文件的“...”就像这样：

```
include_once("fx_server.auth.inc1.php");
...
include_once("fx_server.auth.inc2.php");
```

如果用户能直接连接到 fx_server.auth.inc1.php 或者 fx_server.auth.inc2.php，这可就不太好了，所以要用这些 .htaccess 代码拒绝这些访问。是的，又是正则表达式：

```
<Files ~ "^fx_server[.]auth[.]inc[12][.]php$">
    deny from all
</Files>
```

先来看看后端，上面的附注解释了为什么有六个文件：inc1 和 inc2 包含了大部分代码（它们和第 7 章末尾的代码类似，所以这里不再介绍了），然后其他四个文件和本章前面介绍的三个 auth_test.html 后端文件类似，第四个文件的变化是完全不做任何认证。最后这个文件使我们可以看到由于认证问题的存在会有哪些部分失效，而且也可以展现出当使用 IP 地址作为认证权衡时会发生什么。

fx_server.auth.noauth.php 和 fx_server.auth.apache.php 的代码是一样的（因为对 fx_server.auth.apache.php 来说，是由 Apache 来处理认证，如果用户是非法的，下面这段脚本永远不会被调用到）：

```
<?php
    include_once("fx_server.auth.inc1.php");
    sendHeaders();
    include_once("fx_server.auth.incs.php");
```

（真正的 fx_server.auth.apache.php 代码，除了这些之外，做了一个快速的合理性检查，以确保 Apache 认证正确运转。）

下面是 fx_server.auth.php.php 脚本在 PHP 中处理基本认证的版本：

```
<?php
include_once("fx_server.auth.inc1.php");

$user = @$_SERVER["PHP_AUTH_USER"];
$pw = @$_SERVER["PHP_AUTH_PW"];
$fromDB = '$2a$10$4LLeBta770Y0Z7795j.8' .
    'He/ZCQonnvImXIX0egalzE1MuWiEa6PQa';
if (!password_verify($pw, $fromDB)) {
    header('WWW-Authenticate: Basic realm="SSE Book"');
    header("HTTP/1.0 401 Unauthorized");
    echo "Please authenticate.\n";
    exit;
}

sendHeaders();
```

```
include_once("fx_server.auth.inc2.php");
```

注意 sendHeaders() 是如何在验证之后调用的，如果出现问题，希望返回认证请求头，而不是 SSE 请求头。

最后，下面是最复杂的版本，基于 Cookie 数据做的自定义认证。除了与前面的例子不同之外，它将从 Cookie 或 POST 中接收认证数据：

```
<?php
include_once("fx_server.auth.inc1.php");
sendHeaders();
if (array_key_exists("login", $_COOKIE)) $d = $_COOKIE["login"];
elseif (array_key_exists("login", $_POST)) $d = $_POST["login"];
else {
    sendData(array(
        "action" => "auth",
        "msg" => "The login data is missing. Exiting."
    ));
    exit;
}
if (strpos($d, ",") === false) {
    sendData(array(
        "action" => "auth",
        "msg" => "The login data is invalid. Exiting."
    ));
    exit;
}
list($user, $pw) = explode(",", $d);
$fromDB = '$2a$10$4LLeBta770Y0Z7795j.8' .
    'He/ZCQonnvImXIX0egalzE1MuWiEa6PQa' ;
if (!password_verify($pw, $fromDB)) {
    sendData(array(
        "action" => "auth",
        "msg" => "The login is bad. Exiting."
    ));
    exit;
    }
include_once("fx_server.auth.inc2.php");
```

首先调用 sendHeaders()，这样可以用 sendData() 返回认证失败，它们会以 SSE 消息的方式返回给浏览器。



从浏览器中对 SSE 后端进行故障诊断是令人沮丧的经历。但是，在这里不能像前面几章那样从命令行运行 PHP 脚本，因为需要指定请求头和 Cookie。这种快速测试最好的选择是 curl。下面是测试这三种认证方式的命令（这里假定文件在 http://example.com 的 sse/ 目录下，可以调整一下适配你的安装路径）。

```
curl -uoreilly:test http://example.com/sse/fx_server.auth.apache.php
```

```
curl -uoreilly:test http://example.com/sse/fx_server.auth.php.php
```

```
curl --cookie "login=oreilly,test"
http://example.com/sse/fx_server.auth.custom.php
```

添加 `-v` 参数可以看请求头，或者 `--trace` 查看什么在传送的信息。添加 `-H "Origin: http://127.0.0.1"` 指定一个域。

可以随意填写 Cookie 或用户名：密码来看一下报错信息。

也可以确保这些连接失败：

```
curl http://example.com/sse/fx_server.auth.inc1.php
```

```
curl http://example.com/sse/fx_server.auth.inc2.php
```

现在回到前端，页面就像图 9-1 所示。

图 9-1：fx_client.auth.html 初始化视图

以下是该文件与之前的 fx_client.xhr.html（第 7 章末）和 fx_client.cors.html（本章前部分）的主要区别。

- 一个可以做以下 3 项选择的表单：(1) 采用什么连接技术（SSE、XHR、iframe、长轮询）；(2) 目标 URL（比如，可以修改域名、IP 地址或者在 HTTP 和 HTTPS 之间切换）；(3) 采用什么认证技术。
- 添加了一个无认证技术的选项。
- 会基于 user-agent 检测旧版的 Chrome 和 Safari 浏览器。
- 当使用自定义的方式连接到不同的域，XHR 会采用 POST 数据，而不是使用 Cookie，原生 SSE 也会切换到 XHR 兼容方案来使用 POST。
- 认证失败会被截获并上报。

把这些汇总到一起，就产生了 fx_client.auth.html 这个最长的源文件，但是大部分新代码是表单处理，这里不深入探讨。这里也不会探讨 Chrome/Safari 检测，那只用到了一些正则表达式。

这里要介绍的第一段代码非常简单。自定义认证的代码（fx_server.auth.custom.php）需要上报一个错误时，会通过 SSE 数据流返回。通过设置 `action` 字段为 `"auth"` 来标识。所以，在 `processOneLine()` 中，添加了下面这一段：

```
function processOneLine(s){
    ...
    else if(d.action == "auth"){
        var x = document.getElementById("msg");
        x.innerHTML += "Auth Failure:" + d.msg + "<br/>";
        disconnect();
        }
}
```

对 `disconnect()` 的调用非常重要：这里不想让它一直尝试连接，甚至不想一个长连接机制一直尝试连接。

现在有了一个表单，如果用户在已经有连接在运行时点击其中一个连接按钮会发生什么？有一个叫 `reconnect()` 的新函数就用来处理这种情况：

```
this.reconnect = function (newUrl, newOptions) {
    disconnect();
    url = newUrl;
    for (var key in newOptions)
        options[key] = newOptions[key];
    connect();
}
```

所以，这里首先调用 `disconnect()` 以确保不仅当前连接已关闭，并且所有的计时器也停止。然后设置新 URL，以及任何新的设置，然后尝试带着那些新设置连接到这个新 URL。

从 SSE 到 XHR 的回退兼容通过将下面这段加粗的代码添加到 `startSSE()` 函数来实现：

```
function startEventSource() {
    if (es) {
        es.close();
        es = null;
    }
    if (!isSameDomain()) {
        if (options.post || isOldSafariChrome) {
            startXHR();
            return;
        }
    }
    if (options.post)document.cookie = options.post + "; path=/";
    var u = url;
    if (lastId)u += "lastId="
        + encodeURIComponent(lastId) + "&";
    es = new EventSource(u, { withCredentials: true });
    es.addEventListener("message", function (e) {
        processOneLine(e.data);
    }, false);
    es.addEventListener("error", handleError, false);
}
```

这里的背景是，`options` 对象有一个可选的 `post` 字段，当使用自定义认证时放 `"login=username,`

password"。加粗部分的第一个代码块是说，当连接到一个不同的域并且想要发送 cookie 时它不会工作，所以用 XHR 方案代替。第二部分是说，如果想要发送 cookie 并且要连接到相同的域，就设置一个 cookie。

isOldSafariChrome 是因为没有实现 SSE 的 CORS 的旧浏览器不能跨域，不论是否发送 cookie，所以应该使用 XHR 方案。

第二部分是如何在 startXHR() 中处理 POST：

```
function startXHR() {
    ...
    var ds = null;
    fallback = "xhr=1&t=" + (new Date().getTime());
    if (options.post) {
        xhr.open("POST", url, true);
        xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
        ds = fallback + "&" + options.post;
    }
    else {
        xhr.open("GET", url + fallback, true);
    }
    ...
    xhr.send(ds);
}
```



fx_client.xhr.html 中的代码有这个非常不同，因为它将大部分代码移到一个叫 useXMLHttpRequest() 的辅助函数中，这个函数在 startXHR() 和 startLongPoll() 中都会用到。

所以，当没有设置 options.post，它就和之前的代码一样：xhr 和 t 会在 URL 中发送。但当设置了 options.post，需要设置一个额外的请求头，然后将所有要发送的数据放到 ds 中，它会被传递给 xhr.send(ds)。

就是这样了。尝试一些测试。比如，如果访问，将“Base URL to connect to”修改为“https://example.com/sse/listings/”，或者“http://www1.example.com/sse/listings/”等。然后每个按钮都点一下，看看是否有数据传过来，在 Firebug（或者其他任何你在用的开发者工具）中看一下连接是否是使用 SSE、XHR 或者长轮询，是否是用 GET 或 POST，发送的 cookie 是什么。

## 9.14 未来会有更多一样

本章又长又复杂，如果满足以下两个条件，它本可以相当简单：(1)SSE 标准，以及它的实现，允许像 Ajax 那样设置请求头以及发送 POST 数据；(2) 不存在旧浏览器。

从过去 15 年的经验来看，旧浏览器和浏览器 bug 总是如影随形，我们要时刻准备着与它们做斗争。但是，要处理第一点（SSE 标准的限制），由于我们已经为旧浏览器写了向后兼容方案，我们可以相对容易地处理那些限制。事实上，解决办法就像下面这么简单：

```
if(!isSameDomain()){
    if(options.post || isOldSafariChrome){startXHR();return;}
}
```

服务端推送事件 API 仍然非常新，在接下来的一两年中出现改进也并不稀奇。但即便是现在它就已经非常有用了，我希望你能发现它在你项目里的更多用处。

# 附录A

# SSE标准

写作本书的时候，服务器推送事件的官方标准是一个“W3C 候选推荐标准”。最新的发布版本参见 http:// www.w3.org/TR/eventsource/。

## A.1 W3C候选推荐标准2012.12.11

该版本：

http://www.w3.org/TR/2012/CR-eventsource-20121211/

最新发布版本：

http://www.w3.org/TR/eventsource/

最新编辑草案：

http://dev.w3.org/html5/eventsource/

历史版本：

http://www.w3.org/TR/2012/WD-eventsource-20121023/
http://www.w3.org/TR/2012/WD-eventsource-20120426/
http://www.w3.org/TR/2011/WD-eventsource-20111020/
http://www.w3.org/TR/2011/WD-eventsource-20110310/
http://www.w3.org/TR/2011/WD-eventsource-20110208/
http://www.w3.org/TR/2009/WD-eventsource-20091222/

http://www.w3.org/TR/2009/WD-eventsource-20091029/
http://www.w3.org/TR/2009/WD-eventsource-20090423/

编辑：

Ian Hickson (ian@hixie.ch，Google 公司)。

版权声明：



## A.1.1 摘要

本标准以 DOM 事件的形式，为从服务器接收推送消息定义了打开 HTTP 连接的 API。API 被设计成通过扩展可以和短信推送之类的其他消息推送方案协同工作。

## A.1.2 本文档的状态

本节描述了本文档在发布时的状态。其他文档可以取代本文档。本技术报告的当前 W3C 发布版本和最新修订版列表，参见 http://www.w3.org/TR/ 的 W3C 技术报告索引（W3C technical reports index）。

如果想以 W3C 追踪的方式对本文档进行评论，可以通过 https://www.w3.org/Bugs/Public/describecomponents.cgi?product=WebAppsWG 提交。如果没有帐号，可以使用 http://www.w3.org/TR/eventsource/ 的表单提交反馈。

也可以将反馈意见发邮件至 public-webapps@w3.org（邮件归档：http://lists.w3.org/Archives/Public/public-webapps/，可在该页面订阅邮件列表），或者 whatwg@whatwg.org（订阅：http://lists.whatwg.org/listinfo.cgi/whatwg-whatwg.org，归档：http://lists.whatwg.org/pipermail/whatwg-whatwg.org/）。欢迎反馈。

本标准以及相关标准的修改通知将以如下机制发送：

电子邮件通知修改

Commit-Watchers 邮件列表（完整源码差异）：http://lists.whatwg.org/listinfo.cgi/commit-watchers-whatwg.org

可浏览的所有修改版本控制记录：

有并排的文件差异比较的 CVSWeb 界面：http://dev.w3.org/cvsweb/html5/
有统一文件差异比较的注释摘要：http://html5.org/tools/web-apps-tracker
原始的 Subversion 接口：svn checkout http://svn.whatwg.org/webapps/

W3C 网络应用工作组是负责本标准 W3C 推荐标准追踪进展的 W3C 工作组。本标准是 2012 年 12 月 11 日的候选推荐标准。没有关于 2012 年 10 月 23 日最终草案的注解或 bug。

以候选推荐标准发布并不意味着被 W3C 成员认可。这是一个文档草案，随时可能被其他文档更新、取代或废止。不宜将本文档引用为非进行中的作品。

本文档由一个遵循 2004 年 2 月 5 日 W3C 专利政策（http://www.w3.org/Consortium/Patent-Policy-20040205/）的小组创作。W3C 维护了一个与工作组相关联的公布任何专利的公共列表（http://www.w3.org/2004/01/pp-impl/42538/status），这个页面也包含了公布一个专利的说明。根据 W3C 专利政策第 6 节（http://www.w3.org/Consortium/Patent-Policy-20040205/#sec-Disclosure），任何知道一个专利包含了必要要求（http://www.w3.org/Consortium/Patent-Policy-20040205/#def-essential）的个人，必须公布这些信息。

### 候选推荐标准退出条件

要退出候选推荐（CR）状态，必须满足以下条件。

(1) 必须有至少两个可操作的实现方案通过了本标准测试包（http://w3c-test.org/webapps/ServerSentEvents/tests/）中所有认可的测试用例。一个实现方案必须是可获取（即可以下载）、可分享（即非私有）并且不是试验性的（即面向广大用户）。工作组会决定测试包什么时候达到了可以充分测试可交互性的品质，并且会写出一个实现报告（附属在测试包中）。
(2) 必须在 CR 状态停留至少两个月（也就是在 2013 年 2 月 11 日以后）。这是为了保障有足够的时间发现重大错误。如果实现方案出现比较慢的话，CR 期会延长。

## A.1.3　目录

(9) 无连接推送和其他特性
(10) 垃圾回收
(11) IANA 须知
(12) 参考文献
(13) 致谢

## A.1.4 引言

本节是非规范性的。

为确保服务器通过 HTTP 或专门的服务端推送协议向网页推送数据，本标准引入了 `EventSource`（参见 http://www.w3.org/TR/eventsource/##eventsource）接口。

本接口的使用包含了创建 `EventSource` 对象和注册事件侦听。

```
var source = new
     EventSource('updates.cgi'); source.onmessage = function (event) {
     alert(event.data); };
```

在服务端，脚本（这里指 updates.cgi）用下面的格式，以 text/event-stream 这种 MIME 类型发送消息：

```
data: This is the first message.
    data: This is the second message, it data: has two lines. data: This is
    the third message.
```

可以用不同的事件类型来区分事件。下面这段数据流有两种事件类型：`add` 和 `remove`：

```
event: add data: 73857293 event:
     remove data: 2153 event: add data: 113411
```

处理这段数据流的脚本如下所示（`addHandler` 和 `removeHandler` 是接收事件对象这一个参数的函数）：

```
var source = new
     EventSource('updates.cgi'); source.addEventListener('add', addHandler,
     false); source.addEventListener('remove', removeHandler, false);
```

默认的事件类型是 `message`。

事件流请求可以像普通 HTTP 请求一样用 HTTP 301 和 HTTP 307 重定向。客户端可以在连接关闭时重新连接，可以用 HTTP 204 无内容响应码告知客户端停止重连。

使用本 API，而不是用 `XMLHttpRequest` 或 iframe 仿效，可以在终端实现者和网络运营商能够预先协作的情况下，使客户端更好地利用网络资源。在所有的收益中，这会显著节省便

携设备的耗电量。这会在 A.1.2 节（http://www.w3.org/TR/eventsource/#eventsource-push）中深入探讨。

## A.1.5 一致性要求

本标准中所有的图表、示例以及注释都是非规范性的，每一节也都显著标识为非规范性。本标准其他一切都是规范的。

本文档规范部分的关键字 `MUST`（必须）、`MUST NOT`（不准/禁止）、`REQUIRED`（要求）、`SHOULD`（应该）、`SHOULD NOT`（不应该）、`RECOMMENDED`（推荐）、`MAY`（可能），`OPTIONAL`（可选的）会在 RFC2119 中加以解释。为了方便阅读，这些词再本规范中不会都以大写字母出现。[RFC2119]（http://www.w3.org/TR/eventsource/#refsRFC2119）

算法命令中标出的要求（比如“删除字符串前面的空字符”或者“返回 `false` 并且中止这些步骤”）将会用介绍这个算法的关键字（must、should、may，等等）的意思解释。

有一些一致性要求描述为关于属性、方法和对象的要求。这些要求会被解释为终端相关的要求。

只要结果是一样的，以算法或专门步骤描述的一致性要求可以用任何方式实现。（实际上，本标准所定义的算法是为方便而不是为性能实现的。）

本标准定义的唯一的一致性类是终端。

终端可以在其他不受约束的输入上强加实现特有的限制，比如为防止拒绝服务攻击，为防止耗尽内存，或者为绕开平台特有的限制。

如果某项特性的支持被禁用（比如，为缓解一个安全问题的紧急措施，或者为帮助开发，或者为性能原因），终端必须表现得无论如何都不支持这项特性，并且就像本标准中没有这项特性。比如，如果某个特性是通过一个 Web IDL 接口的属性访问，那这个属性应该从实现了这个接口的对象移除：将这个属性保留在对象上，但使它返回空值或者抛出一个异常是不够的。

### 2.1 依赖

本标准依赖于如下数个其他标准。

HTML

　许多 HTML 的基本概念被本标准所采用。[HTML（http://www.w3.org/TR/eventsource/#refs HTML）]

WebIDL

本标准的 IDL 部分使用了 WebIDL 标准的语法。[WEBIDL]（http://www.w3.org/TR/eventsource/#refsWEBIDL）

WebMessaging

MessageEvent 的定义参见 [WEBMESSAGING]（http://www.w3.org/TR/eventsource/#refsWEBMESSAGING）。

## A.1.6 术语

“一个 Foo 对象”这种句法（Foo 实际上是一个接口）经常用来取代更精确的句法“一个实现了 Foo 接口的对象”。

DOM 这一术语用于表示 Web 应用中可被脚本访问的接口集，不需要表明存在真实 Document 对象或任何其他 DOM Core 标准定义的 Node 对象。[DOMCORE]（http://www.w3.org/TR/eventsource/#refsDOMCORE）

IDL 属性的值被返回称为取值（比如，通过脚本），IDL 属性被赋予一个新值称为设值。

## A.1.7 EventSource接口

```
[Constructor(DOMString url, optional EventSourceInit eventSourceInitDict)]
interface EventSource : EventTarget {
    readonly attribute DOMString url;
    readonly attribute boolean withCredentials;

    // 准备状态
    const unsigned short CONNECTING = 0;
    const unsigned short OPEN = 1;
    const unsigned short CLOSED = 2;
    readonly attribute unsigned short readyState;

    // 网络事件
    attribute EventHandler onopen;
    attribute EventHandler onmessage;
    attribute EventHandler onerror;
    void close();
};

dictionary EventSourceInit {
    boolean withCredentials = false;
};
```

EventSource() 构造函数接收一到两个参数。第一个指定要连接的 URL，第二个以 EventSourceInit（http://www.w3.org/TR/eventsource/##eventsourceinit）字典的格式指定配置

（如果有的话）。当调用 EventSource() 构造函数时，终端必须执行以下步骤。

(1) 解析第一个参数指定的 URL，相对于入口脚本的基准网址（Base URL）。[HTML]（http://www.w3.org/TR/eventsource/#refsHTML）
(2) 如果上一步失败，抛出一个 SyntaxError 异常。
(3) 创建一个新的 EventSource 对象。
(4) 将 CORS 模式设置为匿名。
(5) 如果有第二个参数，并且字典成员 withCredentials 值为 true，则将 CORS 模式设置为使用证书并且初始化新的 EventSource 对象的 withCredentials 属性为 true。
(6) 返回新的 EventSource 对象，并且在后台继续这些步骤（不阻塞脚本执行）。
(7) 用入口脚本的引用来源对解析结果绝对 URL 进行一次潜在允许 CORS 的抓取，模式为 CORS 模式，域为入口脚本的域，如果获得资源，以下面描述的方式处理。



抓取算法（CORS 使用）的定义如下：如果浏览器已经取得给定的绝对 URL 所标识的资源，这个连接可以复用，而不用建立一个新连接。在这种情况下，到目前为止接收的所有消息会立即发送出去。

当脚本的全局对象是一个 Window 对象或一个实现了 WorkerUtils 接口的对象，这个构造函数必须是可见的。

url 属性必须返回绝对 URL，即传给构造函数的 URL 解析结果。

withCredentials 属性必须返回它最后初始化的值。当对象被创建时，它必须被初始化为 false。

readyState 属性代表了连接状态。它可以有下列值。

CONNECTING（数值为 0）

　连接尚未建立，或者已经关闭，并且终端在重连。

OPEN（数值为 1）

　终端有一个打开的连接并且会在接收到事件时派发它们。

CLOSED（数值为 2）

　连接没有打开，终端没有尝试重连。要么是出现了致命错误，要么是调用了 close() 方法。

对象创建时，它的 readyState 必须设置为 CONNECTING(0)。如下处理连接的规则定义了这些值何时改变。

`close()` 方法必须中止本 `EventSource` 对象启动的一切抓取算法实例，并且必须设置 `readyState` 属性值为 `CLOSED`。

下面这些 IDL 属性形式的事件处理程序（以及它们对应的所处理的事件类型），必须被所有实现了 `EventSource` 接口的对象支持。

| 事件处理程序 | 所处理的事件类型 |
| --- | --- |
| `onopen` | `open` |
| `onmessage` | `message` |
| `onerror` | `error` |

作为上面的补充，每个 EventSource 对象都有如下关联

- 以毫秒为单位的重连时间。它的初始值必须由终端定义，取值范围大概为几秒。
- 最后事件 ID 字符串。它的初始值必须是空字符串。

接口现在没有暴露这些值。

## A.1.8　处理模型

参数中指定给 `EventSource`（http://www.w3.org/TR/eventsource/##eventsource）构造函数的资源会在构造函数运行时拉取。

对 HTTP 连接来说，可能会引入 `Accept` 请求头。如果引入了，必须只包含终端支持的事件框架格式（其中一个必须是 text/event-stream（http://www.w3.org/TR/eventsource/##text-event-stream），下文会讲到）。

如果事件源的最后事件 ID 字符串（http://www.w3.org/TR/eventsource/#concept-event-stream-last-event-id）不为空，那么请求中必须包含一个 `Last-Event-ID`（http://www.w3.org/TR/eventsource/##last-event-id）的 HTTP 请求头，它的值为事件源的最后事件 ID 字符串，编码为 UTF-8。

终端应该在请求中使用 `Cache-Control: no-cache` 来为事件源请求绕开缓存。（这个请求头不是一个自定义请求头，所以终端仍然会使用 CORS 简单跨域请求机制。）终端应该在响应中忽略 HTTP 缓存请求头，绝不缓存事件源。

一旦接收到数据，排在网络任务源后面、用以处理数据的任务必须按如下方式执行。

响应码为 HTTP 200 OK，包含一个指定类型为 text/event-stream 的 `Content-Type` 请求头，忽略任何 MIME 类型的参数，必须依下文所述的方式逐行处理（http://www.w3.org/TR/eventsource/#event-stream-interpretation）。

当接收到支持的 MIME 类型的成功响应，终端就开始解析数据流的内容，终端必须广播这

个连接（http://www.w3.org/TR/eventsource/#announce-the-connection）。

网络任务源在该资源（正确的 MIME 类型）抓取算法完成时安置在任务队列里的任务必须触发终端异步重建连接（http://www.w3.org/TR/eventsource/#reestablish-the-connection）。无论连接是正常关闭或异常关闭，这一点都适用于。但它不适用于以下错误情况，除非是显式指定。

响应码为 HTTP 200 OK，但 `Content-Type` 指定了一种不支持的类型，或者根本没有 `Content-Type`，这种情况必须触发终端连接失败（http://www.w3.org/TR/eventsource/#fail-the-connection）。

HTTP 305 Use Proxy、401 Unauthorized 以及 407 Proxy Authentication Required 应该透明地被其他子资源处理。

HTTP 301 Moved Permanently、302 Found、303 See Other 以 及 307 Temporary Redirect 是用抓取和 CORS 算法处理。在 301 重定向的情况中，终端必须也记住新 URL，以便 EventSource 对该资源的后续请求以该对象这些请求的最后一个 301 提供的 URL 开始。

HTTP 500 Internal Server Error、502 Bad Gateway、503 Service Unavailable 以 及 504 Gateway Timeout 响应，以及任何会首先阻止建立连接的网络错误（比如 DNS 错误），必须触发终端异步重建连接。

其他任何没有在这里列出的 HTTP 响应码必须触发终端连接失败。

对于非 HTTP 协议，终端应该以同等方式表现。

当终端要宣布连接，它必须启动这样一个任务，如果 `readyState` 属性的值不为‘CLOSED’，设置 `readyState` 值为 `OPEN`，并在 `EventObject` 对象上触发一个 `open` 事件。

当终端要重建一个连接，必须按以下步骤进行。这些步骤是异步进行的，不是某个任务的一部分（队列中的任务，当然是像普通任务一样并且不是异步的）。

(1) 将运行如下步骤的任务加入队列。
   a. 如果 `readyState` 属性值为 `CLOSED`，终止该任务。
   b. 设置 `readyState` 属性值为 `CONNECTING`。
   c. 在 `EventSource` 对象上触发一个 `error` 事件。

(2) 等待一段与事件源重连时间等长的延时。

(3) 可以选择等待更长时间，特别是当之前的尝试失败，终端会引入一个指数回退延时，以避免对一个可能已经过载的服务器再超载。替代方案是，如果操作系统已经报告了没有网络连接，终端可以等待操作系统通知有连接时再重试。

(4) 如果它还没有运行的话，等到前面提到的任务已经运行。

(5) 将运行如下步骤的任务加入队列。

a. 如果 readyState 属性值不为 CONNECTING，终止这些步骤。

b. 用相同的引用源以及相同的模式和域，执行一个潜在的、允许跨域的、对事件源资源绝对地址的数据抓取，就像被 EventSource() 构造函数触发的原始请求所使用的那样。如果有的话，处理这次获取的资源，就如本节前面所描述的那样。

当终端连接失败，必须在队列中加入一个任务。如果 readyState 属性值不为 CLOSED，设置 readyState 值为 CLOSED 并且在 EventSource 对象上触发一个简单的 error 事件。一旦终端连接失败，它不会尝试重连！

EventSource 对象入队的任何任务的任务源都是远程事件任务源。

## A.1.9 解析事件流

该事件流格式的 MIME 类型是 text/event-stream。

该事件流格式由如下 ABNF stream 产品描述，它的字符集是 Unicode。ABNF

```
stream = [ bom ] *event event =
     *( comment / field ) end-of-line comment = colon *any-char end-of-line
     field = 1*name-char [ colon [ space ] *any-char ] end-of-line
     end-of-line = ( cr lf / cr / lf ) ; characters lf = %x000A ; U+000A LINE
     FEED (LF) cr = %x000D ; U+000D CARRIAGE RETURN (CR) space = %x0020 ;
     U+0020 SPACE colon = %x003A ; U+003A COLON (:) bom = %xFEFF ; U+FEFF
     BYTE ORDER MARK name-char = %x0000-0009 / %x000B-000C / %x000E-0039 /
     %x003B-10FFFF ; a Unicode character other than U+000A LINE FEED (LF), ;
     U+000D CARRIAGE RETURN (CR), or U+003A COLON (:) any-char = %x0000-0009
     / %x000B-000C / %x000E-10FFFF ; a Unicode character other than U+000A
     LINE FEED (LF) ; or U+000D CARRIAGE RETURN (CR)
```

这种格式的事件流必须总是以 UTF-8 编码。RFC3629

行分隔符必须是 U+000D CARRIAGE RETURN U+000A LINE FEED（CRLF）字符对 [ 一个 U+000A LINE FEED（LF）字符 ]，或者是一个单个的 U+000D CARRIAGE RETURN（CR）字符。

由于为这种资源向远程服务器建立连接被料定为是长连接，终端需要确保使用合适的缓冲。特别是当多行的行缓冲被定义为以一个 U+000A LINE FEED（LF）字符结束是安全的，块缓冲或者用不同预期的行结束符的行缓冲会引起事件派发的延迟。

## A.1.10 解释事件流

数据流必须以 UTF-8 编码，并具有错误处理。HTML

一行开始的任何 U+FEFF BYTE ORDER MARK 字符必须忽略（如果有的话）。

数据流必须被逐行解析，一行的结束是以一个 U+000D CARRIAGE RETURN U+000A LINE FEED（CRLF）字符对 [ 一个没有跟在 U+000D CARRIAGE RETURN（CR）之后的 U+000A LINE FEED（LF）字符 ] 和一个没有在 U+000A LINE FEED（LF）之前的 U+000D CARRIAGE RETURN（CR）字符。

当一个数据流被解析，必须有与之关联的数据缓冲、事件类型缓冲和一个最后事件 ID 缓冲。它们的初始值必须是空字符串。

行必须以它们接收的顺序处理，如下所示。

如果该行是空的（空行）触发事件（http://www.w3.org/TR/eventsource/#dispatchMessage），依如下定义。

如果该行以 U+003A COLON 字符（:）开始

忽略这一行。

如果该行包含一个 U+003A COLON（:）字符

收集第一个 U+003A COLON 字符（:）之前的字符，将其作为字段。收集第一个 U+003A COLON 字符（:）之后的字符，将其作为值，如果值以一个 U+0020 SPACE 字符开始，将其从值中移除。依照如下步骤处理字段，将字段作为字段名，将值作为字段值。

另外，字符串不为空但是不包含一个 U+003A COLON 字符（:）

依照如下步骤处理字段，用整行作为字段名，用空字符串作为字段值。

一旦到达文件最后，任何挂起的数据必须废弃（如果文件在事件中间结束，在最后的空行之前，这个未完成的事件不会派发）。

给定字段名和字段值的字段处理步骤取决于字段名，如下列表所示。字段名必须字面匹配，没有大小写嵌套。

如果字段名是“event”

设置事件类型缓冲为字段值。

如果字段名是“data”

将字段值附加到数据缓冲，然后附加一个单个的 U+000A LINE FEED（LF）字符到数据缓冲。

如果字段名是“id”

设置事件 ID 缓冲为字段值。

如果字段名为“retry”

如果字段值仅由 ASCII 数字构成，则将字段值解析为一个十进制整数，然后将其设置为事件流的重连时间。否则，忽略这个字段。

其他

忽略字段。

当终端被要求派发事件，终端必须以如下步骤执行。

(1) 设置事件源的最后事件 ID 字符串为最后事件 ID 缓冲的值。缓冲不会重置，所以事件源的最后事件 ID 会一直是这个值，直到服务端下一次修改。
(2) 如果数据缓冲是一个空字符串，设置数据缓冲和事件类型缓冲为空字符串并且终止这些步骤。
(3) 如果数据缓冲的最后字符是 U+000A LINE FEED（LF），则从数据缓冲中将该字符移除。
(4) 创建一个使用 `MessageEvent` 接口的事件，事件类型为 `message`。它没有冒泡，不可取消，没有默认行为。`data` 属性必须初始化为数据缓冲的值，`origin` 属性必须初始化为事件流最后的 URL 经 Unicode 序列化之后的域（比如重定向之后的 URL），`lastEventId` 属性必须初始化为事件源的最后事件 ID 字符串。该事件是不可信的。
(5) 如果事件类型缓冲有值而不是一个空字符串，将新创建的事件的类型设置为事件类型缓冲的值。
(6) 设置数据缓冲和事件类型缓冲为空字符串。
(7) 将一个任务排入队列，如果 `readyState` 属性值不是 `CLOSED`，派发 `EventSource` 对象新创建的事件。



如果事件没有“id”字段，但是之前的事件确实设置了事件源的最后事件 ID 字符串，那么事件的 `lastEventId` 字段会设置为“id”字段最后的任何值。

下面的事件流，后面一旦是一个空行：

```
data: YHOO data: +2 data: 10
```

会引起 `EventSource` 对象派发一个 `MessageEvent` 接口的 `message` 事件。事件的 `data` 属性会包含 YHOO\n +2\n10 字符串（\n 代表一个新行）。

可能会像下面这样使用：

```
var stocks = new
     EventSource("http://stocks.example.com/ticker.php"); stocks.onmessage =
     function (event) { var data = event.data.split('\n');
     updateStocks(data[0], data[1], data[2]); };
```

updateStocks() 是一个函数，定义如下所示：

```
function updateStocks(symbol,
     delta, value) { ... }
```

或者是类似的定义。

下面的数据流包含了 4 个块。第一块只有一个注释，并不会触发任何事件。第二块有两个名字分别为“data”和“id”的字段，这个块会触发一个事件，带着数据“first event”，然后会设置最后事件 ID 为“1”，以便如果在这一块和下一块之间连接断开，服务端可以发送一个值为“1”的 Last-Event-ID 请求头。第三块触发一个数据为“second event”的事件，并且也有“id”字段，这一次没有值，重置了最后事件 ID 为空字符串（意味着在尝试重连事件中不会发送 Last-Event-ID 请求头）。最终，最后一块只是触发数据为“third event”的事件（有一个前置的空格字符）。注意，最后一个块仍然需要用空行结束，数据流的结束不足以触发最后事件的派发。

```
: test stream data: first event
     id: 1 data:second event id data: third event
```

接下来的数据流触发两个事件：

```
data data data data:
```

第一块触发数据为空字符串的事件，最后一块如果后面跟着一个空行的话也会这样。中间的块触发数据为单个新行字符的事件。最后的块因为没有跟着一个空行而被废弃。

下面的数据流触发两个完全相同的事件：

```
data:test data: test
```

因为冒号后面如果出现空格会被忽略。

## A.1.11　注意事项

据了解，在某些情况下，遗留代理服务器会在短暂的超时之后丢弃 HTTP 连接。为避免这类代理服务器的问题，开发者可以大概每 15 秒发送一个注释行（以 : 字符开始）。

开发者们希望可以将事件源连接彼此关联或者关联到之前有用的规范性文档，他们会发现依赖 IP 地址不可行，因为个别的客户端会有多个 IP 地址（有多个代理服务器），个别的 IP 地址能有多个客户端（共享一个代理服务器）。最好是在使用时将唯一标识符引入文档，然后在连接建立后将标识符作为 URL 的一部分传递。

开发者们也会被告诫 HTTP 分块在协议的可靠性方面有一些意料之外的负面影响。如果可能，支持数据流时应该禁用分块，除非消息频率不会引起问题。

如果打开多个连接同一个站点的页面，并且每个页面有一个连接同一域名的 EventSource 对象，支持 HTTP 单个服务器连接限制的客户端可能会有问题。开发者们可以通过使用相对复杂的机制避免这种问题，即每个连接使用独立的域名，或者允许用户在单页层面启用或禁用 EventSource 功能，或者通过使用一个共享的 Worker 分享单个的 EventSource 对象。WEBWORKERS（www.w3.org/TR/eventsource/#refsWEBWORKERS）

## A.1.12　无连接推送和其他特性

运行在一个受限环境的终端，比如与特定运行商捆绑的手机，可能把连接管理转接给网络代理。在这种情况下，出于一致性的目的，终端会同时包含手机软件和网络代理。

比如，一台移动设备上的浏览器，在已经建立连接后，可能检测到它在一个支持网络上，并且请求网络上的一个代理来接管连接管理。这种情况的时间轴将如下所示。

(1) 浏览器连接到一个远程 HTTP 服务器，并且请求开发者在 EventSource 构造函数中指定的资源。
(2) 服务端发送临时性消息。
(3) 在两次消息之间，浏览器检测出它是闲置的，只有用以保持 TCP 连接活跃的网络活动，于是决定切换到休眠模式以节省电力。
(4) 浏览器与服务器断开连接。
(5) 浏览器联系上了网络上的一个服务，并且请求那个服务，一个“推送代理”，来维护连接。
(6)“推送代理”服务联系远程的 HTTP 服务器，并且请求开发者在 EventSource 构造函数（可能包含一个 Last-Event-ID HTTP 请求头，等等）中指定的资源。
(7) 浏览器允许移动设备进入休眠。
(8) 服务器发送另一条消息。
(9)“推送代理”服务使用一种诸如 OMA 推送的技术将事件传送给移动设备，刚够唤醒来处理事件，然后回到休眠状态。

这可以减少总的数据使用量，因此可以节省大量的电力。

就像实现既有的 API 和本规范定义的 text/event-stream 格式，以及更多如上文所述的分布式方式，其他适用的规范定义的事件框架格式也可支持，本规范没有定义它们应该如何解析或处理。

## A.1.13　垃圾回收

当 EventSource 对象的 readyState 是 CONNECTING，并且该对象注册了一个或多个针对

open、message 或 error 事件的侦听函数时，一定有从调用 EventSource 对象构造函数的 Window 或 WorkerUtils 对象到 EventSource 对象自身的强引用。

当 EventSource 对象的 readyState 是 OPEN，并且该对象注册了一个或多个针对 message 或 error 事件的侦听函数，一定有从调用 EventSource 对象构造函数的 Window 或 WorkerUtils 对象到 EventSource 对象自身的强引用。

当远程事件任务源的 EventSource 对象将一个任务加入队列，一定有从调用 EventSource 对象构造函数的 Window 或 WorkerUtils 对象到 EventSource 对象自身的强引用。

如果终端要强制关闭一个 EventSource 对象（这在 Document 对象永久消失时会发生），终端必须终止 EventSource 对象启动的任何抓取算法实例，并且必须设置 readyState 属性为 CLOSED。

如果 EventSource 对象在它的连接还处于打开状态时被垃圾回收，终端必须终止该 EventSource 对象打开的任何抓取算法实例。



有可能多个 EventSource 对象以及它们的抓取算法共享一个活跃的网络连接，这就是为什么上文描述的是终止抓取算法而不是实际的潜在下载。

## A.1.14　IANA须知

### 1. text/event-stream

这段注册是给社区评审用的，并且会提交到 IESG 评审、审批，并且对 IANA 注册。

类型名称：

　text

子类型名称：

　event-stream

必选参数：

　没有参数。

可选参数：

　charset

　可以提供 charset 参数，参数值必须是 utf-8。该参数没有特别目的，只是考虑兼容遗留服务器。

编码须知：

8 位（总是 UTF-8）。安全须知：来自与内容不同域的事件流会导致信息泄漏。要避免这种情况，要求终端应用 CORS 语法。

CORS 事件流可以压垮终端。终端应该用适当的限制避免因为事件流的信息量过多而耗尽本地资源。

服务器可能会因为引起客户端非常迅速地重连而被压垮。服务器应该用一个 5xx 状态码指示能力极限问题，这会阻止客户端自动重连。

交互性须知：

处理符合规范和不符合规范内容的规则都在本标准中定义。

已发布的标准：

本文档是相关标准。

使用本媒体类型的应用程序：

网络浏览器和使用网络服务的工具。

其他信息。

魔法数字：

没有字节序列可以唯一标识一个事件流。

文件扩展名：

没有推荐给这种类型的专门的文件扩展名。

麦金塔文件类型码：

没有推荐给这种类型的专门的麦金塔文件类型码。

欲知更多信息，请联系本人：

Ian Hickson (ian@hixie.ch)

使用预期：

公用。

使用限制：

本格式预期使用于以 HTTP 或类似协议供应的动态开放式数据流，限定式资源不适用这

种类型。

作者：

Ian Hickson ian@hixie.ch

变更管理：

W3C

片段标识符对 text/event-stream 资源没有意义。

2. `Last-Event-ID`

本节描述了一个注册在永久注册消息头字段里的请求头字段。RFC3864

请求头字段名：

`Last-Event-ID`

试用协议：

http

状态：

标准。

作者 / 变更管理：

W3C

标准文档：

本文档是相关标准。

相关信息：

无。

## A.1.15 参考文献

所有被引用的参考文献都是标准的，除非标记了“非标准”。

[ABNF]

Augmented BNF for Syntax Specifications: ABNF（http://www.ietf.org/rfc/std/std68.txt），D. Crocker, P. Overell. IETF.

[CORS]

Cross-Origin Resource Sharing（http://www.w3.org/TR/cors/）, A. van Kesteren. W3C.

[DOMCORE]

DOM4（http://www.w3.org/TR/dom/）, A. van Kesteren. W3C.

[HTML]

HTML5（http://www.w3.org/TR/dom/）, I. Hickson. W3C.

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels（http://tools.ietf.org/html/rfc2119）, S. Bradner. IETF.

[RFC3629]

UTF-8, a transformation format of ISO 10646（http://tools.ietf.org/html/rfc3629）, F. Yergeau. IETF.

[RFC3864]

Registration Procedures for Message Header Fields（http://tools.ietf.org/html/rfc3864), G. Klyne, M. Nottingham, J. Mogul. IETF.

[WEBIDL]

Web IDL（http://heycam.github.io/webidl/）, C. McCormack. W3C.

[WEBWORKERS]

Web Workers（http://dev.w3.org/html5/workers/）, I. Hickson. W3C.

[WEBMESSAGING]

Web Messaging（http://dev.w3.org/html5/postmsg/）, I. Hickson. W3C.

## A.1.16 致谢

完整的致谢名单，参见 HTML 标准。[HTML]（http://www.w3.org/TR/eventsource/#refs-HTML）

附录B

# 重构：JavaScript全局变量、对象和闭包

你知道用全局变量不好，对吧？一本正经的计算机科学类图书告诉了我们这一点。但它们带来非常多的便利！不必为一个很长的传参列表而浪费时间（或者把这个很长的参数列表重构为单个参数对象，这会使代码量翻倍）。不需担心作用域，它们就在那里（好吧，在JavaScript 以及许多语言中是这样的，在 PHP 中，需要用 `globals` 关键字声明全局变量，或者用 `$_GLOBALS[]`）。当需要修改它们时，不需要担心返回值或是引用参数的问题。所以不使用全局变量有什么好理由吗？测试、便利、封装、副作用，等等。

但是，在数据推送应用的场景中，有一种情况下全局变量会带来麻烦：当需要建立两个甚至更多连接时。



本附录只探讨重构 JavaScript，使其不使用全局变量。之所以将其作为附录，是因为这里介绍的是一种通用的 JavaScript 技术：完全没有专门针对数据推送（当然，示例代码除外）。基本上，作为附录是因为把它作为正文中的附注，内容有点太多了。

## B.1 示例

这里会用一个精简的 SSE 示例。代码将没有有趣的数据，不会有针对旧版浏览器的兼容代码。那些都不会影响哪个方案更好的决策，它只是加了几行代码。

首先来看后端代码：

```
<?php
header("Content-Type: text/event-stream");

function sendData($data) {
echo "data:";
echo json_encode($data) . "\n";
echo "\n"; // 额外的空行
@flush(); @ob_flush();
}

//-------------------------------------
while (true) {

switch (rand(1, 10)) {
    case 1:
        sendData(array("comeBackIn10s" => true));
        exit;
    case 2:
      sendData(array("msg" => "About to sleep 10s"));
      sleep(10); // 强制的长连接延时
      break;
    default:
      sendData(array("t" => date("Y-m-d H:i:s")));
      sleep(1);
      break;
    }
}
```

`while(true)switch(rand(1,10)){...}` 意思着无限循环，并且每次循环随机选择要做什么。80% 的时间都会选中 `default:` 语句，并且只返回一个时间戳。在第 2 章的第一个例子中已经出现过类似代码，所以这里就不再解释代码和 `sendData()` 函数了。

更有趣的是，有 10% 的时间（`case 2:`）它会休眠 10 秒。这是模拟一个死掉的连接：10 秒够用了，因为这里会设置 JavaScript 的长连接超时时间为 5 秒。这里也会返回一条消息，这样当执行到这里时我们就会看到。

`case 1:` 会怎样？会返回一个特别的标记，然后退出。这模拟了 5.4 节中介绍的定期关闭方案。就如这个标记名所显示的那样，想要客户端在 10 秒后重连。

来看一下前端代码，如下所示：

```
    <!doctype html>
    <html>
      <head>
        <meta charset="UTF-8">
        <title>SSE: Basic With Sleep: Globals</title>
</head>
<body>
<pre id="x">Initializing...</pre>
```

```
<script>
    var url = "basic_with_sleep.php";
    var es = null;
    var keepaliveSecs = 5;
    var keepaliveTimer = null;
    function gotActivity() {
        if (keepaliveTimer != null)
            clearTimeout(keepaliveTimer);
        keepaliveTimer = setTimeout(
            connect, keepaliveSecs * 1000);
    }
    function connect() {
        document.getElementById("x").
            innerHTML += "\nIn connect";
        if (es)es.close();
        gotActivity();
        es = new EventSource(url);
        es.addEventListener("message",
            function (e) {processOneLine(e.data);},
                false);
    }
    function processOneLine(s) {
        gotActivity();
        document.getElementById("x").
            innerHTML += "\n" + s;
        var d = JSON.parse(s);
        if (d.comeBackIn10s) {
            if (keepaliveTimer != null)
                clearTimeout(keepaliveTimer);
            if (es)es.close();
            setTimeout(connect, 10 * 1000);
            }
        }

        setTimeout(connect, 100);

        </script>
    </body>
    </html>
```

第 5 章介绍过，全局变量 keepaliveSecs 和 keepaliveTimer 以及 gotActivity() 函数一起协作以保证连接正常。在本书的大部分代码示例中，connect() 函数兼做了 connect() 和 startEventSource() 的工作。这里只是做了一下简化，因为不需要处理向后兼容。processOneLine() 只是输出它正接收的原始 JSON。processOneLine() 的第二部分处理了 comeBackIn10s 消息（这与第 5 章介绍的 temporarilyDisconnect() 函数等效）。

如果你读到这儿之前没有读过第 3 章至第 5 章，并且这让你有些困惑，大可对这段代码具体在做什么放轻松一点。这里要指出的重点是。

- 有 4 个全局变量（一个是参数，其他 3 个是变量）。
- 这 4 个函数每个都用了至少两个全局变量。

- connect() 会从下面 3 个不同的地方调用。
  — 全局代码的初始化调用（实际上延时了 100 毫秒）。
  — 长连接超时以后。
  — 在一个请求返回 10 秒以后。
- 当调用 connect() 时，在创建新连接之前会关闭前一个连接，这是 es（EventSource 对象）作为全局变量的唯一原因。

在浏览器中访问 basic_with_sleep.html，会看到如下输出：

```
Initializing...
In connect
{"t":"2014-02-28 09:46:34"}
{"t":"2014-02-28 09:46:35"}
{"t":"2014-02-28 09:46:36"}
{"msg":"About to sleep 10s"}
In connect
{"t":"2014-02-28 09:46:42"}
{"comeBackIn10s":true}
In connect
{"t":"2014-02-28 09:46:53"}
{"t":"2014-02-28 09:46:54"}
.
.
.
```

当进入休眠时，不会收到任何新数据，所以 5 秒后长连接计时器生效，关闭了旧的连接，并开启了一个新连接，所以会看到 6 秒的间隔。当它说 10 秒后回来时，这里关闭了连接，关闭了长连接计时器，并且礼貌地小睡了 10 秒，所以时间戳里有一个 11 秒的间隔。

## B.2 问题是……

……两个连接。运行本书源码的 basic_with_sleep.two.html。这里不会展示那段代码，因为它太可怕了：有 7 个全局变量，以及 8 个全局函数。写这段代码需要注意力高度集中，即便如此，我还是搞错了，不得不在 Firebug 中调试它们。好吧，你是对的，全局变量不好。



公平起见，basic_with_sleep.two.html 里的代码是非常原始、原生的。用两个带参数的辅助函数可以使它看上去好一点（就像是一个戴着假发和珍娜 - 路易斯· 科尔曼面具的假人，当你凑上去想亲一下时却发现有点不对劲……）。

所以，需要做一些事情。后面会介绍并比较两种 JavaScript 解决方案。

## B.3 JavaScript对象和构造函数

JavaScript 是面向对象的语言，或者说它自称是。如果你觉得争论这个是件有意义的事，大可在自己有空时去做，不是现在，不是这里。示例代码有点像对象，那把它做成一个 JavaScript 对象如何？那 4 个全局变量可以作为成员变量，4 个函数作为成员函数。

关于这个主题有一个很赞的资料，就是 John Resig 和 Bear Bibeault 写的 Secrets of the JavaScript Ninja（Manning，2012）。

来快速回顾一下 JavaScript 对象，但在那之前，来快速回顾一下 JavaScript 函数，特别是 this 变量。函数是 JavaScript 一等对象，意思是它们可以作为参数传递，定义回调函数很容易，它们可以被赋予一些属性。可以给函数传参数。但也有两个隐式传入的参数。一个是 `arguments`，这个参数个数不定的函数很有用。另一个是 `this`，它指向函数的上下文，并且 `this` 总是有定义的，即便函数不是一个类的一部分。当用普通方式调用函数时，`this` 是全局作用域（在浏览器中等同于 `window`）。当函数作为一个对象的成员调用时，`this` 指向这个对象。当函数是一个 DOM 事件回调时，`this` 是触发事件的 DOM 对象。

函数也可以用 `new` 关键字调用，这样是将函数当成一个构造函数。但是在 JavaScript 中，构造函数也等同于其他语言中的 `class` 关键字。它不仅做了初始化，也会描述这个对象里有什么。所以，在构造函数里，`this` 指向新创建的对象，下面是例子：

```
function MyClass(constructorParam) {
    var privateVariable = "hello";

    this.publicVariable = "world";

    var privateFunction = function (a, b) {
        console.log(a + " " + b + constructorParam);
        };

    this.publicFunction = function () {
        privateFunction(
        privateVariable,
        this.publicVariable
        );
      };
    }
```

然后可以像下面这样使用：

```
var x = new MyClass("!");
x.publicFunction();
```

（这段代码会在控制台输出“hello world!”。）

注意，`constructorParam` 和 `privateVariable` 如何表现得像全局变量但却只能在 `MyClass` 的公有和私有成员函数中访问到。完美！

## B.4　用对象的代码

所以，要做一个对象，只需将所有代码都包在一个构造函数中，然后在所有东西前面加上 this.？代码如下所示：

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>SSE: Basic With Sleep: OOP (doesn’t work)</title>
</head>
<body>
<pre id="x">Initializing...</pre>
<script>
function SSE(url, domId) {
this.es = null;
this.keepaliveSecs = 5;
this.keepaliveTimer = null;

this.gotActivity = function () {
    if (this.keepaliveTimer != null)
    clearTimeout(this.keepaliveTimer);
this.keepaliveTimer = setTimeout(
    this.connect, this.keepaliveSecs * 1000);
};

this.connect = function () {
document.getElementById(domId).
  innerHTML += "\nIn connect";
if (this.es)this.es.close();
this.es = new EventSource(url);
this.es.addEventListener(‘message’,
    function (e) {this.processOneLine(e.data); },
    false);
this.gotActivity();
 };

this.processOneLine = function (s) {
this.gotActivity();
document.getElementById(domId).
    innerHTML += "\n" + s;
var d = JSON.parse(s);

if (d.comeBackIn10s) {
    if (this.keepaliveTimer != null)
        clearTimeout(this.keepaliveTimer);
        if (this.es)this.es.close();
        setTimeout(this.connect, 10 * 1000);
        }
    };

    this.connect();
    }
```

```
    setTimeout(function () {new SSE("basic_with_sleep.php", "x"); }, 100);

</script>
</body>
</html>
```

将它保存为 basic_with_sleep.oop1.html，然后在浏览器中访问。呃……什么都没有。Firebug 会报错：“`TypeError: this.processOneLine is not a function`”。噢，是这样的。浏览器到底把 `this.processOneLine = func tion(s){...}` 当成什么意思了呢 ?! 没有比这更像函数的了。肯定是浏览器 bug。

不。问题是 `this` 在那一行是别的意思，它是 `EventSource` 对象的“message”事件处理程序。所以在那个事件处理程序中 `this` 不是指向 SSE 对象，而是指向 `EventSource` 对象。

也许可以聪明一点，把 `processOneLine` 移到 es 上，这样它就能被发现。但是这样的话 `processOneLine` 中所有对 `this` 的引用就不生效了。不，这个方法不对。有一种更简单的方法，在构造函数的上部，用一个叫 self 的私有变量引用到 `this`：

```
function SSE(url,domId){
var self = this;
...
```

其他唯一需要做的修改就是把“message”事件处理程序中的 `this.` 改为 `self.`。没有其他地方需要修改了，只有那里。



事实上，可以把整个类中所有对 `this` 的引用改为 `self`。你甚至可以认为这更加优雅整洁，因此更好。

```
this.es.addEventListener('message',
  function(e){self.processOneLine(e.data);},
  false);
```

basic_with_sleep.oop2.html 是这么做的，试一下会发现这个简单的修改是有效的。耶！面向对象的 JavaScript 挽救了局面。计算机科学家们深鞠一躬，然后写了一个递归函数相互拍背以示安慰。

但还没有完。你不好奇为什么用 `self` 会有效？你不好奇为什么 url 和 domId 不用显式地传入就能被所有函数访问到？

## B.5　JavaScript闭包

这样可行的原因是闭包。不了解闭包也可以用 JavaScript 做很多事，但了解闭包能会让你能量大增。闭包的基本意思是，每次创建一个函数，它就在那一刻给作用域中所有的变量

提供了引用。这里不再讨论过多细节，更深层次的解释参见 John Resig 和 Bear Bibeault 写的 *Secrets of the JavaScript Ninja*（曼宁，2012）。

这意味着在构造函数中，用 `var` 定义一个变量时，这个变量可以被接下来定义的每一个函数访问到。此外，就如上一节介绍的 `self` 例子，它们也可以被这些函数中定义的函数访问到[1]。

看来我们就像公牛闯进了瓷器店一样，深陷于将 `this.` 拍在任何东西前面，其实有更简单的方式。回到原来的代码，有 4 个全局变量和 4 个全局函数。`url` 是参数，所以把它移除，但在其他 3 个全局变量前面加上构造函数的定义，在最后结束构造函数定义，并且调用 `connect()`：

```
    function SSE(url){
    var es = null;
    var keepaliveSecs = 5;
    var keepaliveTimer = null;
    .
    . (functions, untouched)
    .
    connect();
}
```

首先创建一个实例：

```
setTimeout(function(){
    new SSE("basic_with_sleep.php");
  }, 100);
```

如果在浏览器中运行这段代码，它就会生效（参见本书源码的 basic_with_sleep.oop3.html）。所有加在成员变量或函数前面的 `this.` 都不是必须的，`self` 这个别名也不是必须的。

经验教训总结：如果有一些全局变量，以及一些操作它们的全局函数，并且外部访问那些函数有一个唯一的入口，那就把这些整个封装到一个构造函数中。在构造函数的最后调用入口函数，事情就做完了。（如果有其他的外部访问入口，大可添加一些公有函数，用 `this.XXX = function(){...}` 来定义。）

## 两个人的茶和两种茶

要用新构造函数来运行两个连接，并且使它们并排更新，只需做一些快速修改。为它们添加一个独立的 DOM 入口（`id="y"`）。给构造函数添加一个 `domId` 参数。最终，实例化第二个对象（这里的代码用了第二个计时器，在第一个计时器开始之后的两秒开始）。

完整代码（basic_with_sleep.oop3.two.html）如下所示：

注 1：注意，被传来传去的所有这些东西使你的脚本变慢了，这也是要理解闭包的另一个原因。所有这些 `Function` 构造函数，或者用一个函数工厂，是两种避免闭包的方法。

```
<!doctype html>
<html>
<head>
    <meta charset="UTF-8">
    <title>SSE: Basic With Sleep: Simple OOP and Two Instances</title>
    <style>
    pre {float: left;margin: 10px;}
    </style>
</head>
    <body>
    <pre id="x">Initializing X...</pre>
    <pre id="y">Initializing Y...</pre>

    <script>
    function SSE(url, domId) {
    var es = null;
    var keepaliveSecs = 5;
    var keepaliveTimer = null;

    function gotActivity() {
    if (keepaliveTimer != null)
      clearTimeout(keepaliveTimer);
    keepaliveTimer = setTimeout(
      connect, keepaliveSecs * 1000);
    }

    function connect() {
    document.getElementById(domId).
      innerHTML += "\nIn connect";
    if (es)es.close();
    gotActivity();
    es = new EventSource(url);
    es.addEventListener("message",
      function (e) {processOneLine(e.data);},
      false);
     }

    function processOneLine(s) {
    gotActivity();
    document.getElementById(domId).
      innerHTML += "\n" + s;
    var d = JSON.parse(s);

    if (d.comeBackIn10s) {
        if (keepaliveTimer != null)
        clearTimeout(keepaliveTimer);
        if (es)es.close();
        setTimeout(connect, 10 * 1000);
        }
    }

    connect();

    }
```

```
    setTimeout(function () {
        new SSE("basic_with_sleep.php", "x");
        }, 100);
    setTimeout(function () {
        new SSE("basic_with_sleep.php", "y");
        }, 2000);
    </script>
    </body>
</html>
```



切记现代浏览器一般允许向一个域名建立最多 6 个连接（并且这 6 个连接包含了图片请求和 Ajax 请求）。所以如果你试图在前面的测试页添加大量的 SSE 对象，你只能看到最前面的 6 个有更新。

但也不要这样做。无论什么情况都可以用一个 SSE 连接获取所有消息。如果它们用作应用的不同部分，可以用一个 JSON 字段指定消息类型。

不过，对不同服务器的并发连接没有限制，这就是本附录的代码发挥作用的时候了。

# 附录C

# PHP

PHP 被选为本书示例的主要语言，这有多方面的原因。当和 Apache 结合使用时，可使示例代码相当短，只需非常少的框架代码。语法很简单，任何熟悉主流编程语言的人都能看懂。而且它非常符合 SSE 的*既存基础设施优势*（参见 1.5 节），因为很多人的既存基础设施已经建立在 PHP 之上了。

如我所说，我已经尽力使 PHP 代码对任何程序员都是可读的。本附录介绍了一些 PHP 专有的特性。用到这些特性时，相关解释可参考本附录提到的相关小节。

本附录并不是 PHP 的概述。有无数关于这方面的书籍和网站。PHP 手册（http://www.php.net/manual/zh/）是一个非常好的入门资料。如果你在找关于动态网站的书，Robin Nixon 的 *Learning PHP*, *MySQL*, *JavaScript*, *and CSS*（O'Reilly，http://shop.oreilly.com/product/0636920023487.do）很全面，Kevin Tatroe、Peter MacIntyre 和 Rasmus Lerdorf 合著的 *Programming PHP*（O'Reilly，http://shop.oreilly.com/product/0636920012443.do）也是关于动态网页的，主要关注 PHP 语言本身。

## C.1　PHP中的类

PHP（自 PHP5 起）中的类更像 C++、C# 和 Java 中的类，而不太像 JavaScript 中的类。但如果你之前用过任何面向对象语言，应该不难理解本书用到的代码。

类中的每个要素都有一个访问修饰符前缀：`public` 或 `private`。函数的定义以 `function` 关键字开始，没有 `function` 前缀的项就是成员变量。构造函数是对象创建时运行的函数，称为 `__construct()`。所以 fxpair.seconds.php 中封装了一些 `private` 变量、一个初始化那些变

量的构造函数以及一个用那些私有变量和参数做一些事的公有函数。

成员变量（以及成员函数）通过 `$this->` 前缀访问，这和 JavaScript 中用 `this.` 前缀类似。在其他面向对象的语言，比如 C++，用 `this.` 是可选的（虽然有些风格指南建议这样做）。

更多关于 PHP 的面向对象特性，参见手册 http://cn2.php.net/manual/zh/language.oop5.basic.php。

## C.2 随机函数

PHP 有 `rand` 和 `mt_rand` 函数，如何选用它们？本书使用 `mt_rand`。MT 是指 Mersenne Twister（梅森旋转算法），这种算法可以生成更优质的随机数。（有些地方声称 `mt_rand` 明显更快，有些认为 `rand` 和 `mt_rand` 的速度是一样的，这其实取决于操作系统和 PHP 版本。）

用 `mt_srand` 设置 `mt_rand` 的随机种子。每次都设置同样的随机种子可以每次都获取相同的"随机"数序列。这非常利于可复现测试。如果每次都要不同的随机数，就不是特别需要用 `mt_srand` 了，因为 PHP 会初始化随机种子（基于当前服务器时钟）。

## C.3 超全局变量

PHP 有一些超全局变量，可以被所有函数访问到，这些变量提供了解析好的请求信息和服务器环境信息。它们都是关联数组。`$_GET` 是 HTTP GET 数据，`$_POST` 是 HTTP POST 数据，`$_COOKIES` 是什么？你猜吧。`$_SERVER` 会提供关于请求的其他信息，而 `$_ENV` 提供所运行的服务器信息。`$GLOBALS` 可用以访问用户定义的全局变量。

还有 `$_REQUEST`，它合并了所有的 `$_GET`、`$_POST` 和 `$_COOKIES`。要注意的是，一般不鼓励使用 `$_REQUEST`，因为会有 Cookie 数据覆盖表单数据的安全隐患。当你关心的所有变量有效地来自任何 GET、POST 或 Cookie 数据时，才使用且仅使用 `$_REQUEST`。

但是，要注意从 PHP 5.3 起，默认的 php.ini 文件会将 Cookie 数据从 `$_REQUEST` 排除。（参见 http://php.net/request_order。）所以，如果想允许 Cookie 包含在 `$_REQUEST` 中，需要设置 `request_order` 为"CGP"。把"C"放在首位，这样 POST 数据优先级高于 GET 数据，GET 数据优先级高于 Cookie 数据。

## C.4 数据处理

PHP 有一些强大的处理时间和日期的函数。`time` 返回自 1970 年到现在的秒数，这一点在前面介绍过。还有两个前面介绍过的函数，`date()` 和 `gmdate()`。它们分别将 Unix 时间转化为当地时间字符串和 GMT 时间，都有大量可选择的格式（参见 http://php.net/date）。

如果要说哪个 PHP 函数是我在用其他语言时最想念的，那就是 strtotime()。这个函数接收一个字符串格式的日期并返回 Unix 时间（自 1970 年 1 月 1 日起的秒数）。当然它也可以处理标准时间戳格式，比如“2013-12-25 13:25:50”；也可以处理用单词表示的月份，比如“5th December 2013”。但还有更好的！可以指定日期偏移量，所以可以这样写 strtotime("+1 day") 来获取从现在开始 24 小时后的时间戳：可以写“last day of February”，可以写“next month”、“last Thursday”等。

在默认情况下，计算是相对于当前时间的。但是可以通过指定第二个参数使它相对于其他时间。并且第二个参数也可以用 strtotime！下面是找出 2001 年圣诞节前最后一个星期五的例子：

```
$friday = strtotime("last Friday",
    strtotime("2001-12-25"));
echo date("Y-m-d",$friday);
//2001-12-21
```

# C.5 密码

用户密码显然不能以明文存储在数据库中。加密通常也不是好办法：如果密钥被盗，所有的密码都会泄漏。应该散列密码而不是加密。散列是一个单向处理过程：对一个明文密码进行一系列的数学运算来获得一个随机字符串。不能逆向操作，也没有密钥会被盗。但是指定相同的明文密码，总是能用算法获得相同的散列密码。

但是，由于有人制造了越来越快的电脑来黑你，你爷爷的密码散列算法已经不够好了。在过去，md5() 就够用了，如果你拿 salt 和它一起用，你就可以在满是书呆子的公司里面昂首挺胸了。但现在不是了。

自 PHP 5.5.0 之后，安全最佳实践是用 password_hash() 生成密码，用 password_verify() 验证。如果在用早期版本的 PHP，可以将下面这段代码加到脚本[1]最上面（if(!defined(...)) {...} 意思是在 PHP 5.5 之后的版本会忽略这段代码）：

```
if (!defined("PASSWORD_DEFAULT")) {
function password_hash($password) {
$salt = str_replace("+", ".", base64_encode(sha1(time(), true)));
$salt = substr($salt, 0, 22); //We want exactly 22 characters
if (PHP_VERSION_ID < 50307) return crypt($password, '$2a$10$' . $salt);
else return crypt($password, '$2y$10$' . $salt);
}

function password_verify($password, $hash) {
return crypt($password, $hash) === $hash;
}

} //if (!defined('PASSWORD_DEFAULT')) 结束
```

注 1：更全面的版本参见 https://github.com/ircmaxell/password_compat。

PHP 5.3.7 引入了一种新的、更好的散列算法。上面的代码在 PHP 5.3.7 之后用这种算法，否则就使用在那之前更好的选择。`'$2y$10$'` 中的 10 是用以检测有多慢的。在密码散列的怪异世界里，慢是好事。10 是自 PHP 5.5 以来的默认值。它的意思是密码散列可能会占用较多的 CPU 资源。如果要微调这些参数，可以参考 PHP 手册关于这些函数的描述。为防止过时，用一个 `VARCHAR(255)` 字段将密码散列存储到 SQL 数据库中，虽然目前它们总是整整长 60 字符。

## C.6 休眠

PHP 中有两个容易混淆的函数：`sleep()` 和 `usleep()`。前者接收一个表示休眠秒数的整数，后者也接收一个整数，但是作为休眠的微秒数。所以，举个例子来说，`sleep(2)` 和 `usleep(2000000)` 是等效的，都会使脚本休眠 2 秒。但是，如果想要休眠 0.25 秒或者 1.5 秒，唯一的选择是用 `usleep`（分别是 `usleep(250000)` 和 `usleep(1500000)`）。

是时候提一下 `max_execution_time`（一个配置项）和 `set_time_limit()`（重置 `max_execution_time` 的函数）了。0 这个特殊的值表示“一直运行”，这是从命令行运行脚本的默认值。但是，在网络浏览器中，默认值是 30 秒。在 Linux/Mac 系统中，是 30 秒的 CPU 时间。但在 Windows 系统，是用挂钟时间衡量的。对一个流数据服务器来说，30 秒会非常快，除非看着浏览器控制台才会注意到。但 SSE 脚本最终会每 30 秒重连后端。（除非在做密集型计算，在 Linux 上几十分钟甚至数小时都不会注意到。）

解决办法很简单：在脚本的最上面加上下面这行代码：

```
set_time_limit(0);
```

# 封面介绍

本书封面动物是一只短吻针鼹（澳洲针鼹）。只有四种针鼹以及鸭嘴兽是卵生而非胎生的哺乳动物。澳洲针鼹分布在澳大利亚的森林地区（它是那里分布最广泛的“土著”哺乳动物）和新几内亚的部分地区。

澳洲针鼹身长 30 厘米至 45 厘米，皮毛呈棕色，背上覆有奶油色的刺（由角蛋白构成）。短吻针鼹名副其实，它们的吻大约 8 厘米长，比其他种类针鼹的吻短。这个皮质的吻有几个用途：外观呈楔形利于探查昆虫堆；吻中有电感受器，可帮助探测附近的猎物；迷宫一样的骨结构被认为可以帮助凝结蒸发的水蒸气以及使自身降温（由于针鼹没有汗腺）。

针鼹有时被称为带刺的食蚁兽，然而因为它们与真正的食蚁兽没有关系，这个名字也就不再用了。它们以昆虫为食，但主要是蚂蚁和白蚁，它们会挖开昆虫的洞穴，然后用它们又长又粘的舌头捕获猎物。针鼹是挖掘专家，这得益于它们特别的爪子和短而强壮的四肢。除了猎食，它们挖洞也是一种防御手段；如果受到威胁，它们会非常迅速的挖地洞钻进去，将身体卷成一个球，只把锋利的刺露在外面。它们也是游泳健将，可以只把鼻子露出水面，就像一个潜水通气管。

短吻针鼹出现在了澳大利亚 5 分硬币的背面，并且成了经典电子游戏《刺猬索尼克》系列里的角色纳克尔兹。

封面图片来自 Wood 的 *Animate Creation*。

O'REILLY®

# HTML5数据推送应用开发

如今，数据推送技术在网站和Web应用中得到了广泛应用，比如在拍卖网络应用中推送最新出价，在售书网站推送新评论，在在线游戏中推送新高分，推送用户感兴趣的最新微博，等等。

本书是一本简明的数据推送技术指南，作者通过构建一个真实的例子，手把手地向读者展示如何利用HTML5 SSE（Server-Sent Events，服务端推送事件）这项非凡的技术，无需轮询或者用户交互，就可以将最新数据从服务端推送到客户端。

此外，本书还比较了数据推送和WebSocket的区别，阐释了如何使用不同的向后兼容解决方案，将应用的桌面和移动浏览器支持率从60%增加到99%。只要熟悉HTML、HTTP和基本的JavaScript，就可以开始你的学习之旅。

本书主要内容：

- 比较SSE、WebSocket或者数据拉取方案的区别，以便你在解决手头的问题时自如选择
- 开发一个包含后端和前端解决方案的实际SSE应用
- 解决错误处理、系统恢复和其他问题，使应用达到产品水准
- 分析不支持SSE的浏览器的两种向后兼容解决方案
- 处理安全问题，包括认证授权和不允许的域
- 开发在测试驱动SSE设计中有用的实际、可重用的数据
- 学习示例应用中不包含的SSE协议元素

**Darren Cook** 精通多种计算机语言，包括JavaScript、PHP以及C++，拥有20多年软件开发及项目管理经验，涉及金融交易系统、数据可视化工具、世界级公司的网站乃至电子游戏。他开发过类似Twitter的HTTP流数据网络服务系统，还为许多应用写过底层的套接字服务端/客户端协议，构建过使用SSE和WebSocket的应用。

“数据推送是Web应用所涉及的一项关键技术，本书会告诉你如何利用最新的HTML5技术予以实现，并展示各种向后兼容方案的选择。不过在使用之前你仍然要回答这个问题：你的Web应用到底是否需要使用数据推送？当然，在你阅读完本书之后，答案便了然于心。”

——贾铮

百度资深研发工程师

“如果你希望一有最新消息发布，你的Web客户端就立即更新，那么就来学习本书吧。本书展示了利用HTML5和数据推送技术，使你的用户在几乎所有现代平台上及时收到最新消息。”

——**Peter MacIntyre**

Paladin Business Solutions总裁

“HTML5 SSE是响应式动态交互Web前端的未来趋势。本书阐述了如何在客户端和服务端实现SSE。此外，你还将学到PHP的相关知识，以及如何设计高性能、安全的Web应用。”

——**Stuart Woodward**

Hanamaru K.K.高级软件架构师

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

| 分类建议 | 计算机/Web开发/HTML5 |
|---|---|

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



ISBN 978-7-115-37059-4
定价：49.00元

# 看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks